UNISYS

System V Operating System

User's Guide

Volume 2

January 1988

Printed in U S America
UP–13531

Priced Item

UNISYS

System V Operating System

User's Guide

Volume 2

Unisys is a trademark of Unisys Corporation.

January 1988

Printed in U S America
UP–13531

Priced Item

This document is intended for software releases based on AT&T Release 3 of UNIX System V or a subsequent release of the System unless otherwise indicated.

The names, places, and/or events used in this publication are not intended to correspond to any individual, group, or association existing, living, or otherwise. Any similarity or likeness of the names, places and/or events with the names of any individual living or otherwise, or that of any group or association is purely coincidental and unintentional.

NO WARRANTIES OF ANY NATURE ARE EXTENDED BY THIS DOCUMENT. Any product and related material disclosed herein are only furnished pursuant and subject to the terms and conditions of a duly executed Program Product License or Agreement to purchase or lease equipment. The only warranties made by Unisys, if any, with respect to the products described in this document are set forth in such License or Agreement. Unisys cannot accept any financial or other responsibility that may be the result of your use of the information in this document or software material, including direct, indirect, special or consequential damages.

You should be very careful to ensure that the use of this information and/or software material complies with the laws, rules, and regulations of the jurisdictions with respect to which it is used.

The information contained herein is subject to change without notice. Revisions may be issued to advise of such changes and/or additions.

Dataphone and Teletype are registered trademarks of AT&T. DOCUMENTER'S WORKBENCH is a trademark of AT&T. UNIX is a registered trademark of AT&T in the USA and other countries.
HP is a registered trademark of Hewlett-Packard, Inc.
IBM is a registered trademark of International Business Machines Corporation.

Portions of this material are copyrighted © by
AT&T Technologies
and are reprinted with their permission.

7. SHELL TUTORIAL

Chapter 7: Shell Tutorial

Introduction 7-1

Shell Command Language 7-2
- Metacharacters 7-4
  - The Metacharacter That Matches All Characters: the Asterisk (*) 7-4
  - The Metacharacter That Matches One Character: the Question Mark (?) 7-7
  - Using the * or ? to Correct Typing Errors 7-8
  - The Metacharacters That Match One of a Set: Brackets ([ ]) 7-9
- Special Characters 7-10
  - Running a Command in Background: the Ampersand (&) 7-10
  - Executing Commands Sequentially: the Semicolon (;) 7-11
  - Turning Off Special Meanings: the Backslash () 7-12
  - Turning Off Special Meanings: Quotes 7-12
  - Using Quotes to Turn Off the Meaning of a Space 7-13
- Input and Output Redirection 7-15
  - Redirecting Input: the < Sign 7-15
  - Redirecting Output to a File: the > Sign 7-15
  - Appending Output to an Existing File: the > > Symbol 7-16
  - Useful Applications of Output Redirection 7-17
  - Combining Background Mode and Output Redirection 7-20
  - Redirecting Output to a Command: the Pipe (¦) 7-20
  - A Pipeline Using the **cut** and **date** Commands 7-21
  - Substituting Output for an Argument 7-26
- Executing and Terminating Processes 7-26
  - Running Commands at a Later Time With the **batch** and **at** Commands 7-26
  - Obtaining the Status of Running Processes 7-32
  - Terminating Active Processes 7-33
  - Using the **nohup** Command 7-35

Command Language Exercises 7-37

Shell Programming 7-38
Shell Programs 7-39
Creating a Simple Shell Program 7-39
Executing a Shell Program 7-40
Creating a **bin** Directory for Executable Files 7-41
Warnings about Naming Shell Programs 7-42
Variables 7-43
Positional Parameters 7-44
Special Parameters 7-48
Named Variables 7-52
Assigning a Value to a Variable 7-54
Shell Programming Constructs 7-62
Comments 7-62
The here Document 7-63
Using **ed** in a Shell Program 7-65
Return Codes 7-68
Looping 7-69
The Shell's Garbage Can: **/dev/null** 7-75
Conditional Constructs 7-75
Unconditional Control Statements: the **break** and **xontinue** Commands 7-85
Debugging Programs 7-86

Modifying Your Login Environment 7-91
Adding Commands to Your **.profile** 7-91
Setting Terminal Options 7-92
Creating an **rje** Directory 7-94
Using Shell Variables 7-94

Shell Programming Exercises 7-98

Answers To Exercises 7-100
Command Language Exercises 7-100
Shell Programming Exercises 7-101

Introduction

This chapter describes how to use the UNIX* system shell to do routine tasks. For example, it shows you how to use the shell to manage your files, to manipulate file contents, and to group commands together to make programs the shell can execute.

The chapter has two major sections. The first section, "Shell Command Language," covers in detail using the shell as a command interpreter. The second section, "Shell Programming," covers in detail using the shell as a programming language. It tells you how to create, execute, and debug programs made up of commands, variables, and programming constructs like loops and case statements.

The chapter offers many examples. You should login to your UNIX system and recreate the examples as you read the text. As in the other examples in this guide, different type (**bold,** *italic,* and `constant width`) is used to distinguish your input from the UNIX system's output. See the Preface for details.

In addition to the examples, there are exercises at the end of both the "Shell Command Language" and "Shell Programming" sections. The exercises help you better understand the topics discussed. Answers to the exercises are at the end of the chapter.

NOTE: Your UNIX system might not have all commands referenced in this chapter. If you cannot access a command, check with your system administrator.

If you want an overview of how the shell functions as both command interpreter and programming language, see Chapters 1 and 4 before reading this chapter. Also, refer to Appendix E, Summary of Shell Command Language. If you want to learn more advanced concepts in shell programming, you might read *Shell Commands and Programming* (see the *Documentation Roadmap* for information on ordering this manual).

*UNIX is a registered trademark of AT&T in the USA and other countries. Portions of the Unisys System V Operating System are derived from the AT&T V.3 UNIX release.

Shell Command Language

This section introduces commands and, more importantly, some characters with special meanings that let you

- find and manipulate a group of files by using pattern matching
- run a command in the background or at a specified time
- run a group of commands sequentially
- redirect standard input and output from and to files and other commands
- terminate processes

It first covers the characters having special meanings to the shell and then covers the commands and notation for carrying out the tasks listed above. For your convenience, Figure 7-1 summarizes the characters with special meanings discussed in this chapter.

Character	Function
* ? [ ]	metacharacters that provide a shortcut for specifying file names by pattern matching
&	places commands in background mode, leaving your terminal free for other tasks
;	separates multiple commands on one command line
\	turns off the meaning of special characters such as *, ?, [ ], &, ;, >, <, and ¦.
'...'	single quotes turn off the delimiting meaning of a space and the special meaning of all special characters
"..."	double quotes turn off the delimiting meaning of a space and the special meaning of all special characters *except* $ and `
>	redirects output of a command into a file (replaces existing contents)
<	redirects input for a command to come from a file
> >	redirects output of a command to be added to the end of an existing file
¦	creates a pipe of the output of one command to the input of another command
`...`	grave accents allow the output of a command to be used directly as arguments on a command line
$	used with positional parameters and user-defined variables; also used as the default shell prompt symbol

Figure 7-1: Characters with Special Meanings in the Shell Language

Metacharacters

Metacharacters, a subset of the special characters, represent other characters. They are sometimes called wild cards, because they are like the joker in card games that can be used for any card. The metacharacters ***** (asterisk), **?** (question mark), and **[ ]** (brackets) are discussed here.

These characters are used to match file names or parts of file names, thereby simplifying the task of specifying files or groups of files as command arguments. (The files whose names match the patterns formed from these metacharacters must already exist.) This is known as file name expansion. For example, you may want to refer to all file names containing the letter "a", all file names consisting of five letters, and so on.

The Metacharacter That Matches All Characters: the Asterisk (*)

The asterisk (*) matches any string of characters, including a null (empty) string. You can use the * to specify a full or partial file name. The ***** alone refers to all the file and directory names in the current directory. To see the effect of the *, try it as an argument to the **echo**(1) command. Type:

echo *<CR>

The **echo** command displays its arguments on your screen. Notice that the system response to **echo *** is a listing of all the file names in your current directory. However, the file names are displayed horizontally rather than in vertical columns such as those produced by the **ls** command.

Figure 7-2 summarizes the syntax and capabilities of the **echo** command.

Command Recap **echo** - write any arguments to the output		
command	*options*	*arguments*
echo	none	any character(s)
Description:	**echo** writes arguments, which are separated by blanks and ended with <**CR**>, to the output.	
Remarks:	In shell programming, **echo** is used to issue instructions, to redirect words or data into a file, and to pipe data into a command. All these uses will be discussed later in this chapter.	

Figure 7-2: Summary of the **echo** Command

CAUTION: The * is a powerful character. For example, if you type **rm** * you will erase all the files in your current directory. Be very careful how you use it!

For another example, say you have written several reports and have named them **report**, **report1**, **report1a**, **report1b.01**, **report25**, and **report316**. By typing **report1*** you can refer to all files that are part of report1, collectively. To find out how many reports you have written, you can use the **ls** command to list all files that begin with the string "report," as shown in the following example.

```
$ ls report* <CR>
report
report1
report1a
report1b.01
report25
report316
$
```

The * matches any characters after the string "report," including no letters at all. Notice that * matches the files in numerical and alphabetical order. A quick and easy way to print the contents of your report files in order on your screen is by typing the following command:

pr report* <CR>

Now try another exercise. Choose a character that all the file names in your current directory have in common, such as a lower case "a". Then request a listing of those files by referring to that character. For example, if you choose a lower case "a", type the following command line:

ls *a* <CR>

The system responds by printing the names of all the files in your current directory that contain a lower case "a".

The * can represent characters in any part of the file name. For example, if you know that several files have their first and last letters in common, you can request a list of them on that basis. For such a request, your command line might look like this:

ls F*E <CR>

The system response will be a list of file names that begin with F, end with E, and are in the following order:

F123E
FATE
FE
Fig3.4E

The order is determined by the ASCII sort sequence: (1) numbers; (2) upper case letters; (3) lower case letters.

The Metacharacter That Matches One Character: the Question Mark (?)

The question mark (**?**) matches any single character of a file name. Let's say you have written several chapters in a book that has twelve chapters, and you want a list of those you have finished through Chapter 9. Use the **ls** command with the **?** to list all chapters that begin with the string "chapter" and end with any single character, as shown below:

```
$ ls chapter?<CR>
chapter1
chapter2
chapter5
chapter9
$
```

The system responds by printing a list of all file names that match.

Although **?** matches any one character, you can use it more than once in a file name. To list the rest of the chapters in your book, type:

ls chapter??<CR>

Of course, if you want to list all the chapters in the current directory, use the *****:

ls chapter*

Using the * or ? to Correct Typing Errors

Suppose you use the **mv**(1) command to move a file, and you make an error and enter a character in the file name that is not printed on your screen. The system incorporates this non-printing character into the name of your file and subsequently requires it as part of the file name. If you do not include this character when you enter the file name on a command line, you get an error message. You can use ***** or **?** to match the file name with the non-printing character and rename it to the correct name.

Try the following example.

1. Make a very short file called **trial**.
2. Type: **mv trial trial<^g>1<CR>**

   (Remember, to type **<^g>** you must hold down the CONTROL key and press the g key.)
3. Type: **ls trial1<CR>**

   The system will respond with an error message:

   ```
   $ ls trial1<CR>
   trial1: no such file or directory
   $
   ```
4. Type: **ls trial?1<CR>**

   The system will respond with the file name **trial1** (including the non-printing character), verifying that this file exists. Use the **?** again to correct the file name.

   ```
   $ mv trial?1 trial1<CR>
   $ ls trial1<CR>
   trial1
   $
   ```

The Metacharacters That Match One of a Set: Brackets ([ ])

Use brackets ([ ]) when you want the shell to match any one of several possible characters that may appear in one position in the file name. For example, if you include **[crf]** as part of a file name pattern, the shell will look for file names that have the letter "c", the letter "r", or the letter "f" in the specified position, as the following example shows.

```
$ ls [crf]at<CR>
cat
fat
rat
$
```

This command displays all file names that begin with the letter "c", "r", or "f" and end with the letters "at". Characters that can be grouped within brackets in this way are collectively called a "character class".

Brackets can also be used to specify a range of characters, whether numbers or letters. For example, if you specify

chapter[1-5]

the shell will match any files named **chapter1** through **chapter5**. This is an easy way to handle only a few chapters at a time.

Try the **pr** command with an argument in brackets:

$ **pr chapter[2-4]<CR>**

This command will print the contents of **chapter2**, **chapter3**, and **chapter4**, in that order, on your terminal.

A character class may also specify a range of letters. If you specify **[A-Z]**, the shell will look only for upper case letters; if **[a-z]**, only lower case letters.

The uses of the metacharacters are summarized in Figure 7-3. Try out the metacharacters on the files in your current directory.

Character	Function
*	matches any string of characters, including an empty (null) string
?	matches any single character
[ ]	matches one of the sequence of characters specified within the brackets
[–]	matches one of the range of characters specified

Figure 7-3: Summary of Metacharacters

Special Characters

The shell language has other special characters that perform a variety of useful functions. Some of these additional special characters are discussed in this section; others are described in the next section, "Input and Output Redirection."

Running a Command in Background: the Ampersand (&)

Some shell commands take considerable time to execute. The ampersand (&) is used to execute commands in background mode, thus freeing your terminal for other tasks. The general format for running a command in background mode is

command **&<CR>**

NOTE: You should not run interactive shell commands, for example **read** (see "Using the **read** Command" in this chapter), in the background.

In the example below, the shell is performing a long search in background mode. Specifically, the **grep**(1) command is searching for the string "delinquent" in the file **accounts**.) Notice the **&** is the last character of the command line:

```
$ grep delinquent accounts &<CR>
21940
$
```

When you run a command in the background, the UNIX system outputs a process number; 21940 is the process number in the example. You can use this number to stop the execution of a background command. (Stopping the execution of processes is discussed in the "Executing and Terminating Processes" section.) The prompt on the last line means the terminal is free and waiting for your commands; **grep** has started running in background.

Running a command in background affects only the availability of your terminal; it does not affect the output of the command. Whether or not a command is run in background, it prints its output on your terminal screen, unless you redirect it to a file. (See "Redirecting Output," later in this chapter, for details.)

If you want a command to continue running in background after you log off, you can submit it with the **nohup**(1) command. (This is discussed in "Using the **nohup** Command," later in this chapter.)

Executing Commands Sequentially: the Semicolon (;)

You can type two or more commands on one line as long as each pair is separated by a semicolon (;) , as follows:

command1; *command2*; *command3*<CR>

The UNIX system executes the commands in the order that they appear in the line and prints all output on the screen. This process is called sequential execution.

Try this exercise to see how the ; works. First, type

cd; pwd; ls<CR>

The shell executes these commands sequentially:

1. **cd** changes your location to your login directory
2. **pwd** prints the full path name of your current directory
3. **ls** lists the files in your current directory

If you do not want the system's responses to these commands to appear on your screen, refer to "Redirecting Output" for instructions.

Turning Off Special Meanings: the Backslash (\)

The shell interprets the backslash (\) as an escape character that allows you to turn off any special meaning of the character immediately after it. To see how this works, try the following exercise. Create a two-line file called **trial** that contains the following text:

```
The all * game
was held in Summit.
```

Use the **grep** command to search for the asterisk in the file, as shown in the following example:

```
$ grep \* trial<CR>
The all * game
$
```

The **grep** command finds the * in the text and displays the line in which it appears. Without the \, the * would be a metacharacter to the shell and would match all file names in the current directory.

Turning Off Special Meanings: Quotes

Another way to escape the meaning of a special character is to use quotation marks. Single quotes ('...') turn off the special meaning of any character. Double quotes ("...") turn off the special meaning of all characters except **$** and ` (grave accent), which retain their special meanings within double quotes. An advantage of using quotes is that numerous special characters can be

enclosed in the quotes; this can be more concise than using the backslash.

For example, if your file named **trial** also contained the line

```
He really wondered why? Why???
```

you could use the **grep** command to match the line with the three question marks as follows:

```
$ grep '???' trial<CR>
He really wondered why? Why???
$
```

If you had instead entered the command

```
grep ??? trial<CR>
```

the three question marks would have been used as shell metacharacters and matched all file names of length three.

Using Quotes to Turn Off the Meaning of a Space

A common use of quotes as escape characters is for turning off the special meaning of the blank space. The shell interprets a space on a command line as a delimiter between the arguments of a command. Both single and double quotes allow you to escape that meaning.

For example, to locate two or more words that appear together in text, make the words a single argument (to the **grep** command) by enclosing them in quotes. To find the two words "The all" in your file **trial**, enter the following command line:

```
$ grep ´The all´ trial<CR>
The all * game
$
```

grep finds the string "The all" and prints the line that contains it. What would happen if you did not put quotes around that string?

The ability to escape the special meaning of a space is especially helpful when you are using the **banner**(1) command. This command prints a message across a terminal screen in large, poster size letters.

To execute **banner**, specify a message consisting of one or more arguments (in this case usually words), separated on the command line by spaces. The **banner** will use these spaces to delimit the arguments and print each argument on a separate line.

To print more than one argument on the same line, enclose the words, together, in double quotes. For example, to send a birthday greeting to another user, type:

banner happy birthday to you<CR>

The command prints your message as a four-line banner. Now print the same message as a three-line banner. Type:

banner happy birthday "to you"<CR>

Notice that the words "to" and "you" now appear on the same line. The space between them has lost its meaning as a delimiter.

Figure 7-4 summarizes the syntax and capabilities of the **banner** command.

Command Recap **banner** - make posters		
command	*options*	*arguments*
banner	none	characters
Description:	**banner** displays up to ten characters in large letters	
Remarks:	Later in this chapter you will learn how to redirect the **banner** command into a file to be used as a poster.	

Figure 7-4: Summary of the **banner** Command

Input and Output Redirection

In the UNIX system, some commands expect to receive their input from the keyboard (standard input) and most commands display their output at the terminal (standard output). However, the UNIX system lets you reassign the standard input and output to other files and programs. This is known as redirection. With redirection, you can tell the shell to

- take its input from a file rather than the keyboard
- send its output to file rather than the terminal
- use a program as the source of data for another program

You use a set of operators, the less than sign (<), the greater than sign (>), two greater than signs (> >), and the pipe (¦) to redirect input and output.

Redirecting Input: the < Sign

To redirect input, specify a file name after a less than sign (<) on a command line:

command < *file* <**CR**>

For example, assume that you want use the **mail**(1) command (described in Chapter 8) to send a message to another user with the login **colleague** and that you already have the message in a file named **report**. You can avoid retyping the message by specifying the file name as the source of input:

mail colleague < **report** <**CR**>

Redirecting Output to a File: the > Sign

To redirect output, specify a file name after the greater than sign (>) on a command line:

command > *file* <**CR**>

CAUTION: If you redirect output to a file that already exists, the output of your command will overwrite the contents of the existing file.

Before redirecting the output of a command to a particular file, make sure that a file by that name does not already exist, unless you do not mind losing it. Because the shell does not allow you to have two files of the same name in a directory, it will overwrite the contents of the existing file with the output of your command if you redirect the output to a file with the existing file's name. The shell does not warn you about overwriting the original file.

To make sure there is no file with the name you plan to use, run the **ls** command, specifying your proposed file name as an argument. If a file with that name exists, **ls** will list it; if not, you will receive a message that the file was not found in the current directory. For example, checking for the existence of the files **temp** and **junk** would give you the following output.

```
$ ls temp<CR>
temp
$ ls junk<CR>
junk:  no such file or directory
$
```

This means you can name your new output file **junk**, but you cannot name it **temp** unless you no longer want the contents of the existing **temp** file.

Appending Output to an Existing File: the > > Symbol

To keep from destroying an existing file, you can also use the double redirection symbol (>>), as follows:

command > > *file*<**CR**>

This appends the output of a command to the end of the file *file*. If *file* does not exist, it is created when you use the > > symbol this way.

The following example shows how to append the output of the **cat** command to an existing file. First, the **cat** command is first executed on both files without output redirection to show their respective contents. Then the contents of **trial2** are added after the last line of **trial1** by executing the **cat** command on **trial2** and redirecting the output to **trial1**.

```
$ cat trial1 <CR>
This is the first line of trial1.
Hello.
This is the last line of trial1.
$
$ cat trial2 <CR>
This is the beginning of trial2.
Hello.
This is the end of trial2.
$
$ cat trial2 > > trial1 <CR>
$ cat trial1 <CR>
This is the first line of trial1.
Hello.
This is the last line of trial1.
This is the beginning of trial2.
Hello.
This is the end of trial2.
$
```

Useful Applications of Output Redirection

Redirecting output is useful when you do not want it to appear on your screen immediately or when you want to save it. Output redirection is also especially useful when you run commands that perform clerical chores on text files. Two such commands are **spell** and **sort**.

The spell *Command*

The **spell** program compares every word in a file against its internal vocabulary list and prints a list of all potential misspellings on the screen. If **spell** does not have a listing for a word (such as a person's name), it will report that as a misspelling, too.

Running **spell** on a lengthy text file can take a long time and may produce a list of misspellings that is too long to fit on your screen. **spell** prints all its output at once; if it does not fit on the screen, the command scrolls it continuously off the top until it has all been displayed. A long list of misspellings will roll off your screen quickly and may be difficult to read.

You can avoid this problem by redirecting the output of **spell** to a file. In the following example, **spell** searches a file named **memo** and places a list of misspelled words in a file named **misspell**:

$ **spell memo > misspell**<CR>

Figure 7-5 summarizes the syntax and capabilities of the **spell** command.

Command Recap **spell -** find spelling errors		
command	*options*	*arguments*
spell	available*	*file*
Description:	**spell** collects words from a specified file or files and looks them up in a spelling list. Words that are not on the spelling list are displayed on your terminal.	
Options:	**spell** has several options, including one for checking British spellings.	
Remarks:	The list of misspelled words can be redirected into a file.	

* **See the spell**(1) manual page in the *User's Reference Manual* for all available options and an explanation of their capabilities.

Figure 7-5: Summary of the **spell** Command

The sort *Command*

The **sort** command arranges the lines of a specified file in alphabetical order (see Chapter 3 for details). Because users generally want to keep a file that has been alphabetized, output redirection greatly enhances the value of this command.

Be careful to choose a new name for the file that will receive the output of the **sort** command (the alphabetized list). When **sort** is executed, the shell first empties the file that will accept the redirected output. Then it performs the sort and places the output in the blank file. If you type

sort list > list<CR>

the shell will empty **list** and then sort nothing into **list**.

Combining Background Mode and Output Redirection

Running a command in background does not affect the command's output; unless it is redirected, output is always printed on the terminal screen. If you are using your terminal to perform other tasks while a command runs in background, you will be interrupted when the command displays its output on your screen. However, if you redirect that output to a file, you can work undisturbed.

For example, in the "Special Characters" section you learned how to execute the **grep** command in background with **&**. Now suppose you want to find occurrences of the word "test" in a file named **schedule**. Run the **grep** command in background and redirect its output to a file called **testfile**:

$ **grep test schedule > testfile &<CR>**

You can then use your terminal for other work and examine **testfile** when you have finished it.

Redirecting Output to a Command: the Pipe (¦)

The ¦ character is called a pipe. Pipes are powerful tools that allow you to take the output of one command and use it as input for another command without creating temporary files. A multiple command line created in this way is called a pipeline.

The general format for a pipeline is:

command1 ¦ *command2* ¦ *command3*... **<CR>**

The output of *command1* is used as the input of *command2*. The output of *command2* is then used as the input for *command3*.

To understand the efficiency and power of a pipeline, consider the contrast between two methods that achieve the same results.

- To use the input/output redirection method, run one command and redirect its output to a temporary file. Then run a second command that takes the contents of the

temporary file as its input. Finally, remove the temporary file after the second command has finished running.

- To use the pipeline method, run one command and pipe its output directly into a second command.

For example, say you want to mail a happy birthday message in a banner to the owner of the login **david**. Doing this without a pipeline is a three-step procedure. You must

1. Enter the **banner** command and redirect its output to a temporary file:

   banner happy birthday > message.tmp

2. Enter the **mail** command using **message.tmp** as its input:

   mail david < message.tmp

3. Remove the temporary file:

   rm message.tmp

However, by using a pipeline you can do this in one step:

banner happy birthday ¦ mail david<CR>

A Pipeline Using the cut and date Commands

The **cut** and **date** commands provide a good example of how pipelines can increase the versatility of individual commands. The **cut** command allows you to extract part of each line in a file. It looks for characters in a specified part of the line and prints them. To specify a position in a line, use the **−c** option and identify the part of the file you want by the numbers of the spaces it occupies on the line, counting from the left-hand margin.

For example, say you want to display only the dates from a file called **birthdays**. The file contains the following list:

```
Anne      12/26
Klaus     7/4
Mary      10/18
Peter     11/9
Nandy     4/23
Sam       8/12
```

The birthdays appear between the ninth and thirteenth spaces on each line. To display them, type:

cut –c9-13 birthdays<CR>

The output is shown below:

```
12/26
7/4
10/18
11/9
4/23
8/12
```

Figure 7-6 summarizes the syntax and capabilities of the **cut** command.

Command Recap		
cut - cut out selected fields from each line of a file		
command	options	arguments
cut	**–c**list **–f**list [**–d**]	*file*
Description:	**cut** extracts columns from a table or fields from each line of a file	
Options:	**–c** lists the number of character positions from the left. A range of numbers such as characters 1-9 can be specified by **–c1-9** **–f** lists the field number from the left separated by a delimiter described by **–d**. **–d** gives the field delimiter for **–f**. The default is a space. If the delimiter is a colon, this would be specified by **–d :** .	
Remarks:	If you find the **cut** command useful, you may also want to use the **paste** command and the **split** command.	

Figure 7-6: Summary of the **cut** Command

The **cut** command is usually executed on a file. However, piping makes it possible to run this command on the output of other commands, too. This is useful if you want only part of the information generated by another command. For example, you may want to have the time printed. The **date** command prints the day of the week, date, and time, as follows:

```
$ date<CR>
Sat Dec 27 13:12:32 EST 1986
```

Notice that the time is given between the twelfth and nineteenth spaces of the line. You can display the time (without the date) by piping the output of **date** into **cut**, specifying spaces **12-19** with the **–c** option. Your command line and its output will look like this:

```
$ date ¦ cut -c12-19<CR>
13:14:56
```

Figure 7-7 summarizes the syntax and capabilities of the **date** command.

Command Recap **date** - display the date and time		
command	options	arguments
date	**%m%d%y*** **%H%%M%S**	available*
Description:	**date** displays the current date and time on your terminal	
Options:	% followed by **m** (for month), **d** (for day), **y** (for year), **H** (for hour), **M** (for month), and **S** (for second) will echo these back to your terminal. You can add explanations such as: **date ´%H:%M is the time´**	
Remarks:	If you are working on a small computer system of which you are both a user and the system administrator, you may be allowed to set the date and time using optional arguments to the **date** command. Check your reference manual for details. When working in a multiuser environment, the arguments are available only to the system administrator.	

Figure 7-7: Summary of the **date** Command

* See the **date**(1) manual page in the *User's Reference Manual* for all available options and an explanation of their capabilities.

Substituting Output for an Argument

The output of any command may be captured and used as arguments on a command line. This is done by enclosing the command in grave accents (`...`) and placing it on the command line in the position where the output should be treated as arguments. This is known as command substitution.

For example, you can substitute the output of the **date** and **cut** pipeline command used previously for the argument in a **banner** printout by typing the following command line:

$ banner `date ¦ cut -c12-19` <CR>

Notice the results: the system prints a banner with the current time.

The "Shell Programming" section in this chapter shows you how you can also use the output of a command line as the value of a variable.

Executing and Terminating Processes

This section discusses the following topics:

- how to schedule commands to run at a later time by using the **batch** or **at** command
- how to obtain the status of active processes
- how to terminate active processes
- how to keep background processes running after you have logged off

Running Commands at a Later Time With the batch and at Commands

The **batch** and **at** commands allow you to specify a command or sequence of commands to be run at a later time. With the **batch** command, the system determines when the commands run; with the **at** command, you determine when the commands run. Both commands expect input from standard input (the terminal);

the list of commands entered as input from the terminal must be ended by pressing **<^d>** (control-d).

The **batch** command is useful if you are running a process or shell program that uses a large amount of system time. The **batch** command submits a batch job (containing the commands to be executed) to the system. The job is put in a queue, and runs when the system load falls to an acceptable level. This frees the system to respond rapidly to other input and is a courtesy to other users.

The general format for **batch** is:

batch <CR>
first command **<CR>**
.
.
.
last command **<CR>**
<^d>

If there is only one command to be run with **batch**, you can enter it as follows:

batch *command_line* **<CR>**
<^d>

The next example uses **batch** to execute the **grep** command at a convenient time. Here **grep** searches all files in the current directory and redirects the output to the file **dol.file**.

```
$ batch grep dollar * > dol-file<CR>
<^d>
job 155223141.b at Sun Dec 7 11:14:54 1986
$
```

After you submit a job with **batch**, the system responds with a job number, date, and time. This job number is not the same as the

process number that the system generates when you run a command in the background.

Figure 7-8 summarizes the syntax and capabilities of the **batch** Command.

Command Recap **batch** - execute commands at a later time		
command	*options*	*input*
batch	none	*command_lines*
Description:	**batch** submits a batch job, which is placed in a queue and executed when the load on the system falls to an acceptable level.	
Remarks:	The list of commands must end with a <^**d**> (control-d).	

Figure 7-8: Summary of the **batch** Command

The **at** command allows you to specify an exact time to execute the commands. The general format for the **at** command is

at *time* <**CR**>
first command <**CR**>
.
.
.
last command <**CR**>
<^**d**>

The *time* argument consists of the time of day and, if the date is not today, the date.

The following example shows how to use the **at** command to mail a happy birthday banner to login **emily** on her birthday:

```
$ at 8:15am Feb 27<CR>
banner happy birthday ¦ mail emily<CR>
<^d>
job 453400603.a at Thurs Feb 27 08:15:00 1986
$
```

Notice that the **at** command, like the **batch** command, responds with the job number, date, and time.

If you decide you do not want to execute the commands currently waiting in a **batch** or **at** job queue, you can erase those jobs by using the **–r** option of the **at** command with the job number. The general format is

at –r *jobnumber*<**CR**>

Try erasing the previous **at** job for the happy birthday banner. Type in:

at –r 453400603.a<**CR**>

If you have forgotten the job number, the **at –l** command will give you a list of the current jobs in the **batch** or **at** queue, as the following screen shows:

```
$ at -l<CR>
user = mylogin 168302040.a at Sat Nov 29 13:00:00 1986
user = mylogin 453400603.a at Fri Feb 27 08:15:00 1987
$"
```

Notice that the system displays the job number and the time the job will run.

Using the **at** command, mail yourself the file **memo** at noon, to tell you it is lunch time. (You must redirect the file into **mail** unless you use the "here document," described in the "Shell Programming" section.) Then try the **at** command with the **-l** option:

```
$ at 12:00pm<CR>
mail mylogin < memo<CR>
<^d>
job 263131754.a at Jun 30 12:00:00 1986
$
$ at -l<CR>
user = mylogin 263131754.a at Jun 30 12:00:00 1986
$
```

Figure 7-9 summarizes the syntax and capabilities of the **at** command.

Command Recap

at - execute commands at a specified time

command	*options*	*arguments*
at	**-r** **-l**	*time (date)* *jobnumber*

Description: **Executes commands at the time specified. You can use between one and four digits, and am or pm to show the time. To specify the date, give a month name followed by the number for the day. You do not need to enter a date if you want your job to run the same day. See the** **at**(1) manual page in the *User's Reference Manual* for other default times.

Options: The **–r** option with the job number removes previously scheduled jobs.

The **–l** option (no arguments) reports the job number and status of all scheduled **at** and **batch** jobs.

Remarks: Examples of how to specify times and dates with the **at** command:

at 08:15am Feb 27
at 5:14pm Sept 24

Figure 7-9: Summary of the **at** Command

Obtaining the Status of Running Processes

The **ps** command gives you the status of all the processes you are currently running. For example, you can use the **ps** command to show the status of all processes that you run in the background using **&** (described in the earlier section "Special Characters").

The next section, "Terminating Active Processes," discusses how you can use the PID (process identification) number to stop a command from executing. A PID is a number from 1 to 30,000 that the UNIX system assigns to each active process.

In the following example, **grep** is run in the background, and then the **ps** command is issued. The system responds with the process identification (PID) and the terminal identification (TTY) number. It also gives the cumulative execution time for each process (TIME), and the name of the command that is being executed (COMMAND).

```
$ grep word * > temp &<CR>
28223
$
$ ps<CR>
PID          TTY  TIME COMMAND
28124        tty100:00   sh
28223        tty100:04   grep
28224        tty100:04   ps
$
```

Notice that the system reports a PID number for the **grep** command, as well as for the other processes that are running: the **ps** command itself, and the **sh** (shell) command that runs while you are logged in. The shell program **sh** interprets the shell commands and is discussed in Chapters 1 and 4.

Figure 7-10 summarizes the syntax and capabilities of the **ps** command.

Command Recap **ps** - report process status		
command	*options*	*arguments*
ps	several*	none
Description:	**ps** displays information about active processes.	
Options:	Several. If none are specified, **ps** displays the status of all active processes you are running.	
Remarks:	Gives you the PID (process ID). This is needed to **kill** a process (stop the process from executing).	

* **See the ps**(1) manual page in the *User's Reference Manual* for all available options and an explanation of their capabilities.

Figure 7-10: Summary of the **ps** Command

Terminating Active Processes

The **kill** command is used to terminate active shell processes. The general format for the **kill** command is

kill *PID* <**CR**>

You can use the **kill** command to terminate processes that are running in background. Note that you cannot terminate background processes by pressing the BREAK or DELETE key.

The following example shows how you can terminate the **grep** command that you started executing in background in the previous example.

```
$ kill 28223<CR>
28223 Terminated
$
```

Notice the system responds with a message and a $ prompt, showing that the process has been killed. If the system cannot find the PID number you specify, it responds with an error message:

```
kill:28223:No such process
```

Figure 7-11 summarizes the syntax and capabilities of the **kill** command.

Command Recap **kill** - terminate a process		
command	*options*	*arguments*
kill	available*	*job number* or *PID*
Description:	**kill** terminates the process specified by the PID number.	

* **See the kill**(1) manual page in the *User's Reference Manual* for all available options and an explanation of their capabilities.

Figure 7-11: Summary of the **kill** Command

Using the nohup Command

All processes are killed when you log off. If you want a background process to continue running after you log off, you must use the **nohup** command to submit that background command.

To execute the **nohup** command, follow this format:

nohup *command* **&<CR>**

Notice that you place the **nohup** command before the command you intend to run as a background process.

For example, say you want the **grep** command to search all the files in the current directory for the string "word" and redirect the output to a file called **word.list**, and you wish to log off immediately afterward. Type the command line as follows:

nohup grep word * > word.list & <CR>

You can terminate the **nohup** command by using the **kill** command. Figure 7-12 summarizes the syntax and capabilities of the **nohup** command.

Command Recap		
nohup - prevents interrupt of command execution by hang ups		
command	*options*	*arguments*
nohup	*none*	*command line*
Description:	**Executes a command line, even if you hang up or quit the system.**	

Figure 7-12: Summary of the **nohup** Command

Now that you have mastered these basic shell commands and notations, use them in your shell programs! The exercises that follow will help you practice using shell command language. The answers to the exercises are at the end of the chapter.

Command Language Exercises

1-1. What happens if you use an * at the beginning of a file name? Try to list some of the files in a directory using the * with the last letter of one of your file names.

1-2. Try the following two commands; enter them as follows:

cat[0-9]* <CR>
echo * <CR>

1-3. Is it acceptable to use a **?** at the beginning or in the middle of a file name generation? Try it.

1-4. Do you have any files that begin with a number? Can you list them without listing the other files in your directory? Can you list only those files that begin with a lower case letter between a and m? (Hint: use a range of numbers or letters in **[ ]**).

1-5. Is it acceptable to place a command in background mode on a line that is executing several other commands sequentially? Try it. What happens? (Hint: use **;** and **&**.) Can the command in background mode be placed in any position on the command line? Try placing it in various positions. Experiment with each new character that you learn to see the full power of the character.

1-6. Redirect the output of **pwd** and **ls** into a file by using the following command line:

cd; pwd; ls; ed trial <CR>

To redirect both commands to the same file, use the **>>** (append) sign for the second redirection. If you do not, you will wipe out the information from the **pwd** command.

1-7. Instead of cutting the time out of the **date** response, redirect the date, without the time, into **banner**. What is the only part you need to change in the time command line?

banner 'date ¦ cut -c12-19' <CR>

Shell Programming

You can use the shell to create programs – new commands. Such programs are also called "shell procedures." This section tells you how to create and execute shell programs using commands, variables, positional parameters, return codes, and basic programming control structures.

The examples of shell programs in this section are shown two ways. First, the **cat** command is used in a screen to display the contents of a file containing a shell program:

```
$ cat testfile <CR>
first command
     .
     .
     .
last command
$
```

Second, the results of executing the shell program appear after a command line:

```
$ testfile <CR>
program_output
$
```

You should be familiar with an editor before you try to create shell programs. Refer to the tutorials in Chapter 5 (for the **ed** editor) and Chapter 6 (for the **vi** editor).

Shell Programs

Creating a Simple Shell Program

We will begin by creating a simple shell program that will do the following tasks, in order.

- print the current directory
- list the contents of that directory
- display this message on your terminal: "This is the end of the shell" program.

Create a file called **dl** (short for directory list) using your editor of choice, and enter the following:

pwd <CR>
ls <CR>
echo This is the end of the shell program. <CR>

Now write and quit the file. You have just created a shell program! You can **cat** the file to display its contents, as the following screen shows:

```
$ cat dl <CR>
pwd
ls
echo This is the end of the shell program.
$
```

Executing a Shell Program

One way to execute a shell program is to use the **sh** command. Type:

sh dl<CR>

The **dl** command is executed by **sh**, and the path name of the current directory is printed first, then the list of files in the current directory, and finally, the comment `This is the end of the shell program`. The **sh** command provides a good way to test your shell program to make sure it works.

If **dl** is a useful command, you can use the **chmod** command to make it an executable file; then you can type **dl** by itself to execute the command it contains. The following example shows how to use the **chmod** command to make a file executable and then run the **ls −l** command to verify the changes you have made in the permissions.

```
$ chmod ux dl<CR>
$ ls -l<CR>
total 2
-rw-------      1  login login   3661  Nov  2   10:28 mbox
-rwx------      1  login login     48  Nov 15   10:50 dl
$
```

Notice that **chmod** turns on permission to execute (**+x**) for the user (**u**). Now **dl** is an executable program. Try to execute it. Type:

dl<CR>

You get the same results as before, when you entered **sh dl** to execute it. For further details about the **chmod** command, see Chapter 3.

Creating a bin Directory for Executable Files

To make your shell programs accessible from all your directories, you can make a **bin** directory from your login directory and move the shell files to your **bin**.

You must also set your shell variable **PATH** to include your **bin** directory:

PATH = $PATH:$HOME/bin

See "Variables" and "Using Shell Variables" in this chapter for more information about **PATH**. For more advanced information, refer to the manual *Shell Commands and Programming*.

The following example will remind you which commands are necessary. In this example, **dl** is in the login directory. Type these command lines:

cd < CR >
mkdir bin < CR >
mv dl bin/dl < CR >

Move to the **bin** directory and type the **ls -l** command. Does **dl** still have execute permission?

Now move to a directory other than the login directory, and type the following command:

dl < CR >

What happened?

Figure 7-13 summarizes your new shell program, **dl**.

Shell Program Recap **dl** - display directory path & directory contents (user defined)	
command	*arguments*
dl	none
Description:	**dl** displays the output of the shell command **pwd** and **ls**.

Figure 7-13: Summary of the **dl** Shell Program

It is possible to give the bin directory another name; if you do so, you need to change your shell variable **PATH** again.

Warnings about Naming Shell Programs

You can give your shell program any appropriate file name. However, you should not give your program the same name as a system command. If you do, the system will execute your command instead of the system command. For example, if you had named your **dl** program **mv**, each time you tried to move a file, the system would have executed your directory list program instead of **mv**.

Another problem can occur if you name the **dl** file **ls**, and then try to execute the file. You would create an infinite loop, since your program executes the **ls** command. After some time, the system would give you the following error message:

```
Too many processes, cannot fork
```

What happened? You typed in your new command, **ls**. The shell read and executed the **pwd** command. Then it read the **ls** command in your program and tried to execute your **ls** command. This formed an infinite loop.

UNIX system designers wisely set a limit on how many times an infinite loop can execute. One way to keep this from happening is to give the path name for the system's **ls** command, **/bin/ls**, when you write your own shell program.

The following **ls** shell program would work:

```
$ cat ls<CR>
pwd
/bin/ls
echo This is the end of the shell program
```

If you name your command **ls**, then you can only execute the system **ls** command by using its full path name, **/bin/ls**.

Variables

Variables are the basic data objects shell programs manipulate, other than files. Here we discuss three types of variables and how you can use them:

- positional parameters
- special parameters
- named variables

Positional Parameters

A positional parameter is a variable within a shell program whose value is set from an argument specified on the command line invoking the program. Positional parameters are numbered and are referred to with a preceding **$**: **$1**, **$2**, **$3**, and so on.

A shell program may reference up to nine positional parameters. If a shell program is invoked on with a command line that appears like this:

shell.prog pp1 pp2 pp3 pp4 pp5 pp6 pp7 pp8 pp9 < CR >

then positional parameter **$1** within the program will be assigned the value **pp1**, positional parameter **$2** within the program will be assigned the value **pp2**, and so on, when the shell program is invoked.

Create a file called **pp** (short for positional parameters) to practice positional parameter substitution. Then enter the **echo** commands shown in the following screen. Enter the command lines so that running the **cat** command on your completed file will produce the following output:

```
$ cat pp<CR>
echo  The first positional parameter is: $1<CR>
echo  The second positional parameter is: $2<CR>
echo  The third positional parameter is: $3<CR>
echo  The fourth positional parameter is: $4<CR>
$
```

If you execute this shell program with the arguments **one**, **two**, **three**, and **four**, you will obtain the following results (first you must make the shell program **pp** executable using the **chmod** command):

```
$ chmod u+x pp<CR>
$
$ pp one two three four<CR>
The first positional parameter is: one
The second positional parameter is: two
The third positional parameter is: three
The fourth positional parameter is: four
$
```

The following screen shows the shell program **bbday**, which mails a greeting to the login entered in the command line:

```
$ cat bbday<CR>
banner happy birthday | mail $1
```

Try sending yourself a birthday greeting. If your login name is **sue**, your command line will be:

bbday sue<CR>

Figure 7-14 summarizes the syntax and capabilities of the **bbday** shell program.

Shell Program Recap **bbday** - mail a banner birthday greeting (user defined)	
command	*arguments*
bbday	*login*
Description:	**bbday** mails the message happy birthday, in poster-sized letters, to the specified login.

Figure 7-14: Summary of the **bbday** Command

The who command lists all users currently logged in on the system. How can you make a simple shell program called **whoson**, that will tell you if the owner of a particular login is currently working on the system?

Type the following command line into a file called **whoson**:

who ¦ grep $1 <CR>

The **who** command lists all current system users, and **grep** searches the output of the **who** command for a line containing the string contained as a value in the positional parameter $1.

Now try using your login as the argument for the new program **whoson**. For example, say your login is **sue**. When you issue the **whoson** command, the shell program substitutes **sue** for the parameter $1 in your program and executes as if it were:

who ¦ grep sue <CR>

The output is shown on the following screen:

```
$ whoson sue<CR>
sue    tty26       Jan 24 13:35
$
```

If the owner of the specified login is not currently working on the system, **grep** fails and the **whoson** prints no output.

Figure 7-15 summarizes the syntax and capabilities of the **whoson** command.

Shell Program Recap **whoson** - display login info if user is logged in (user defined)	
command	*arguments*
whoson	*login*
Description:	If a user is on the system, **whoson** displays the user's login, the TTY number, and the time and date the user logged in.

Figure 7-15: Summary of the **whoson** Command

The shell allows a command line to contains 128 arguments. However, a shell program is restricted to referencing nine positional parameters, **$1** through **$9**, at a given time. This restriction can be worked around using the **shift** command, described in the manual *Shell Commands and Programming*. The special parameter **$***, described in the next section, can also be used to access the

values of all command line arguments.

Special Parameters

$# This parameter, when referenced within a shell program, contains the number of arguments with which the shell program was invoked. Its value can be used anywhere within the shell program.

Enter the command line shown in the following screen in an executable shell program called **get.num**. Then run the **cat** command on the file:

```
$ cat get.num <CR>
echo The number of arguments is: $#
$
```

The program simply displays the number of arguments with which it is invoked. For example:

```
$ get.num test out this program <CR>
The number of arguments is: 4
$
```

Figure 7-16 summarizes the **get.num** shell program.

Shell Program Recap	
get.num - count and display number of arguments (user defined)	
command	*arguments*
get.num	*(character_string)*
Description:	**get.num** counts the number of arguments given to the command and then displays the total.
Remarks:	This command demonstrates the special parameter **$#**.

Figure 7-16: Summary of the **get.num** Shell Program

$* This special parameter, when referenced within a shell program, contains a string with all the arguments with which the shell program was invoked, starting with the first. You are not restricted to nine parameters as with the positional parameters **$1** through **$9**..

You can write a simple shell program to demonstrate **$***. Create a shell program called **show.param** that will **echo** all the parameters. Use the command line shown in the following completed file:

```
$ cat show.param <CR>
echo The parameters for this command are: $*
$
```

show.param will echo all the arguments you give to the command. Make **show.param** executable and try it out, using these parameters:

Hello. How are you?

```
$ show.param Hello. How are you? <CR>
The parameters for this command are: Hello.  How are you?
$
```

Notice that **show.param** echoes `Hello. How are you?` Now try **show.param** using more than nine arguments:

```
$ show.param one two 3 4 5 six 7 8 9 10 11<CR>
The parameters for this command are: one two 3 4 5 six
7 8 9 10 11 $
```

Once again, **show.param** echoes all the arguments you give. The **$*** parameter can be useful if you use file name expansion to specify arguments to the shell command.

Use the file name expansion feature with your **show.param** command. For example, say you have several files in your directory named for chapters of a book: **chap1**, **chap2**, and so on, through **chap7**. **show.param** will print a list of all those files.

```
$ show.param chap?<CR>
The parameters for this command are: chap1 chap2 chap3
chap4 chap5 chap6 chap7
$
```

Figure 7-17 summarizes the **show.param** shell program.

Shell Program Recap	
show.param - display all positional parameters (user defined)	
command	*arguments*
show.param	(any positional parameters)
Description:	**show.param** displays all the parameters.
Remarks:	If the parameters are file name generations, the command will display each of those file names.

Figure 7-17: Summary of the **show.param** Shell Program

Named Variables

Another form of variable that you can use within a shell program is a named variable. You assign values to named variables yourself. The format for assigning a value to a named variable is

named_variable = *value* < **CR** >

Notice that there are no spaces on either side of the = sign.

In the following example, **var1** is a named variable, and **myname** is the value or character string assigned to that variable:

var1 = **myname** < **CR** >

A **$** is used in front of a variable name in a shell program to reference the value of that variable. Using the example above, the reference **$var1** tells the shell to substitute the value **myname** (assigned to **var1**), for any occurrence of the character string **$var1**.

The first character of a variable name must be a letter or an underscore. The rest of the name can be composed of letters, underscores, and digits. As in shell program file names, it is not advisable to use a shell command name as a variable name. Also, the shell has reserved some variable names you should not use for your variables. A brief explanation of these reserved shell variable names follows:

- **CDPATH** defines the search path for the **cd** command.
- **HOME** is the default variable for the **cd** command (home directory).
- **IFS** defines the internal field separators (normally the space, the tab, and the carriage return).
- **LOGNAME** is your login name.
- **MAIL** names the file that contains your electronic mail.
- **PATH** determines the search path used by the shell to find commands.
- **PS1** defines the primary prompt (default is **$**).
- **PS2** defines the secondary prompt (default is >).
- **TERM** identifies your terminal type. It is important to set this variable if you are editing with **vi**.
- **TERMINFO** identifies the directory to be searched for information about your terminal, for example, its screen size.
- **TZ** defines the time zone (default is **EST5EDT**).

Many of these variables are explained in "Modifying Your Login Environment" later in this chapter. You can also read more about them on the **sh**(1) manual page in the *User's Reference Manual*.

You can see the value of these variables in your shell in two ways. First, you can type

echo $*variable_name*

The system outputs the value of *variable_name*. Second, you can use the **env**(1) command to print out the value of all defined variables in the shell. To do this, type **env** on a line by itself; the system outputs a list of the variable names and values.

Assigning a Value to a Variable

If you edit with **vi**, you know you can set the **TERM** variable by entering the following command line:

TERM = *terminal_name* < **CR** >

This is the simplest way to assign a value to a variable.

There are several other ways to do this:

- Use the **read** command to assign input to the variable.
- Redirect the output of a command into a variable by using command substitution with grave accents (` ... `).
- Assign a positional parameter to the variable.

The following sections discuss each of these methods in detail.

Using the read *Command*

The **read** command used within a shell program allows you to prompt the user of the program for the values of variables. The general format for the **read** command is:

read *variable* < **CR** >

The values assigned by **read** to *variable* will be substituted for $*variable* wherever it is used in the program. If a program executes the **echo** command just before the **read** command, the program can display directions such as `Type in ...` . The **read** command will wait until you type a character string, followed by a RETURN key, and then make that string the value of the variable.

The following example shows how to write a simple shell program called **num.please** to keep track of your telephone numbers. This program uses the following commands for the purposes specified:

echo	to prompt you for a person's last name
read	to assign the input value to the variable **name**
grep	to search the file **list** for this variable

Your finished program should look like the one displayed here:

```
$ cat num.please<CR>
echo Type in the last name:
read name
grep $name list
$
```

Create a file called **list** that contains several last names and phone numbers. Then try running **num.please**.

The next example is a program called **mknum**, which creates a list. **mknum** includes the following commands for the purposes shown.

- **echo** prompts for a person's name
- **read** assigns the person's name to the variable *name*
- **echo** asks for the person's number
- **read** assigns the telephone number to the variable *num*
- **echo** adds the values of the variables *name* and *num* to the file **list**

If you want the output of the **echo** command to be added to the end of **list**, you must use >> to redirect it. If you use >, **list** will contain only the last phone number you added.

Running the **cat** command on **mknum** displays the program's contents. When your program looks like this, you will be ready to make it executable (with the **chmod** command):

```
$ cat mknum<CR>
echo Type in name
read name
echo Type in number
read num
echo $name $num >> list
$ chmod u+x mknum<CR>
$
```

Try out the new programs for your phone list. In the next example, **mknum** creates a new listing for Mr. Niceguy. Then **num.please** gives you Mr. Niceguy's phone number:

```
$ mknum<CR>
Type in the name
Mr. Niceguy<CR>
Type in the number
668-0007<CR>
$ num.please<CR>
Type in last name
Niceguy<CR>
Mr. Niceguy 668-0007
$
```

Notice that the variable **name** accepts both **Mr.** and **Niceguy** as the value.

Figures 7-18 and 7-19 summarize the **mknum** and **num.please** shell programs, respectively.

Shell Program Recap **mknum** - place name and number on a phone list	
command	*arguments*
mknum	(interactive)
Description:	Asks you for the name and number of a person and adds that name and number to your phone list.
Remarks:	This is an interactive command.

Figure 7-18: Summary of the **mknum** Shell Program

Shell Program Recap **num.please** - display a person's name and number	
command	*arguments*
num.please	(interactive)
Description:	Asks you for a person's last name, and then displays the person's full name and telephone number.
Remarks:	This is an interactive command.

Figure 7-19: Summary of the **num.please** Shell Program

Substituting Command Output for the Value of a Variable

You can substitute a command's output for the value of a variable by using *command substitution*. This has the following format:

variable = `*command*` **<CR>**

The output from *command* becomes the value of *variable*.

In one of the previous examples on piping, the **date** command was piped into the **cut** command to get the correct time. That command line was the following:

date | cut -c12-19<CR>

You can put this in a simple shell program called **t** that will give you the time.

```
$ cat t<CR>
time=`date | cut -c12-19`
echo The time is: $time
$
```

Remember there are no spaces on either side of the equal sign. Make the file executable, and you will have a program that gives you the time:

```
$ chmod u+x t<CR>
$ t<CR>
The time is: 10:36
$
```

Figure 7-20 summarizes your **t** program.

Shell Program Recap **t** - display the correct time	
command	*arguments*
t	none
Description:	**t** gives you the correct time in hours and minutes.

Figure 7-20: Summary of the **t** Shell Program

Assigning Values with Positional Parameters

You can assign a positional parameter to a named parameter by using the following format:

var1 = **$1** < **CR** >

The next example is a simple program called simp.p that assigns a positional parameter to a variable. The following screen shows the commands in **simp.p**:

```
$ cat simp.p<CR>
var1=$1
echo $var1
$
```

Of course, you can also assign the output of a command that uses positional parameters to a variable, as follows:

person=`who ¦ grep $1`<CR>

In the next example, the program **log.time** keeps track of your **whoson** program results. The output of **whoson** is assigned to the variable **person**, and added to the file **login.file** with the **echo** command. The last **echo** displays the value of **$person**, which is the same as the output from the **whoson** command:

```
$ cat log.time<CR>
person=`who ¦ grep $1`
echo $person >> login.file
echo $person
$
```

The system response to **log.time** is shown in the following screen:

```
$ log.time maryann <CR>
maryann     tty61          Apr 11 10:26
$
```

Figure 7-21 summarizes the **log.time** shell program.

Shell Program Recap **log.time** - log and display a specified login (user defined)	
command	*arguments*
log.time	*login*
Description:	If the specified login is currently on the system, **log.time** places the line of information from the **who** command into the file **login.file** and then displays that line of information on your terminal.

Figure 7-21: Summary of the **log.time** Shell Program

Shell Programming Constructs

The shell programming language has several constructs that give added flexibility to your programs:

- Comments let you document a program's function.
- The "here document" allows you to include within the shell program itself lines to be redirected to be the input to some command in the shell program.
- The **exit** command lets you terminate a program at a point other than the end of the program and use return codes.
- The looping constructs, **for** and **while**, allow a program to iterate through groups of commands in a loop.
- The conditional control commands, **if** and **case**, execute a group of commands only if a particular set of conditions is met.
- The **break** command allows a program to exit unconditionally from a loop.

Comments

You can place comments in a shell program in two ways. All text on a line following a # (pound) sign is ignored by the shell. The # sign can be at the beginning of a line, in which case the comment uses the entire line, or it can occur after a command, in which case the command is executed but the remainder of the line is ignored. The end of a line always ends a comment. The general format for a comment line is

#comment < **CR** >

For example, a program that contains the following lines will ignore them when it is executed:

```
# This program sends a generic birthday greeting.<CR>
# This program needs a login as<CR>
# the positional parameter.<CR>
```

Comments are useful for documenting a program's function and

should be included in any program you write.

The here Document

A "here document" allows you to place into a shell program lines that are redirected to be the input of a command in that program. It is a way to provide input to a command in a shell program without needing to use a separate file. The notation consists of the redirection symbol << and a delimiter that specifies the beginning and end of the lines of input. The delimiter can be one character or a string of characters; the **!** is often used.

Figure 7-22 shows the general format for a here document.

command < < *delimiter* < **CR** >
...*input lines*... < **CR** >
delimiter < **CR** >

Figure 7-22: Format of a Here Document

In the next example, the program **gbday** uses a here document to send a generic birthday greeting by redirecting lines of input into the **mail** command:

```
$ cat gbday<CR>
mail $1 <<!
Best wishes to you on your birthday.
!
$
```

When you use this command, you must specify the recipient's login as the argument to the command. The input included with the use of the here document is:

```
Best wishes to you on your birthday
```

For example, to send this greeting to the owner of login **mary**, type:

```
$ gbday mary<CR>
```

Login **mary** will receive your greeting the next time she reads her mail messages:

```
$ mail<CR>
From mylogin Wed May 14 14:31 CDT 1986
Best wishes to you on your birthday
$
```

Figure 7-23 summarizes the format and capabilities of the **gbday** command.

Shell Program Recap **gbday** - send a generic birthday greeting (user defined)	
command	*arguments*
gbday	*login*
Description:	**gbday** sends a generic birthday greeting to the owner of the login specified in the argument.

Figure 7-23: Summary of the **gbday** Command

Using ed in a Shell Program

The here document offers a convenient and useful way to use **ed** in a shell script. For example, suppose you want to make a shell program that will enter the **ed** editor, make a global substitution to a file, write the file, and then quit **ed**. The following screen shows the contents of a program called **ch.text** which does these tasks.

```
$ cat ch.text<CR>
echo Type in the file name.
read file1
echo Type in the exact text to be changed.
read old_text
echo Type in the exact new text to replace the above.
read new_text
ed - $file1 <<!
g/$old_text/s//$new_text/g
w
q
!
$
```

Notice the - (minus) option to the **ed** command. This option prevents the character count from being displayed on the screen. Notice, also, the format of the **ed** command for global substitution:

g/*old_text*/*s*//*new_text*/**g**<**CR**>

The program uses three variables: *file1*, *old_text*, and *new_text*. When the program is run, it uses the **read** command to obtain the values of these variables. The variables provide the following information:

file the name of the file to be edited

old_text the exact text to be changed

new_text the new text

Once the variables are entered in the program, the here document redirects the global substitution, the write command, and the quit command into the **ed** command. Try the new **ch.text** command. The following screen shows sample responses to the program prompts:

```
$ ch.text<CR>
Type in the filename.
memo<CR>
Type in the exact text to be changed.
Dear John:<CR>
Type in the exact new text to replace the above.
To whom it may concern:<CR>
$ cat memo<CR>
To whom it may concern:
$
```

Notice that by running the **cat** command on the changed file, you could examine the results of the global substitution.

Figure 7-24 summarizes the format and capabilities of the **ch.text** command.

Shell Program Recap **ch.text** - change text in a file	
command	*arguments*
ch.text	(interactive)
Description:	Replaces text in a file with new text.
Remarks:	This shell program is interactive. It will prompt you to type in the arguments.

Figure 7-24: Summary of the **ch.text** Command

If you want to become more familiar with **ed**, see Chapter 5, "Line Editor Tutorial **(ed)**." The stream editor **sed** can also be used in shell programming. You can find more information on the **sed** editor in the *Editing Guide* or *Editing Utilities Guide*.

Return Codes

Most shell commands issue return codes that indicate whether the command executed properly. By convention, if the value returned is 0 (zero) than the command executed properly; any other value indicates that it did not. The return code is not printed automatically, but is available as the value of the shell special parameter **$?**.

Checking Return Codes

After executing a command interactively, you can see its return code by typing

echo $?

Consider the following example:

```
$ cat hi
This is file hi.
$ echo $?
0
$ cat hello
cat: cannot open hello
$ echo $?
2
$
```

In the first case, the file **hi** exists in your directory and has read permission for you. The **cat** command behaves as expected and outputs the contents of the file. It exits with a return code of 0, which you can see using the parameter **$?**. In the second case, the file either does not exist or does not have read permission for you. The **cat** command prints a diagnostic message and exits with

a return code of 2.

Using Return Codes With the exit *Command*

A shell program normally terminates when the last command in the file is executed. However, you can use the **exit** command to terminate a program at some other point. Perhaps more importantly, you can also use the **exit** command to issue return codes for a shell program. For more information about **exit**, see the **exit**(2) manual page in the *Programmer's Reference Manual* or *Shell Commands and Programming*.

Looping

In the previous examples in this chapter, the commands in shell programs have been executed in sequence. The **for** and **while** looping constructs allow a program to execute a command or sequence of commands several times.

The for *Loop*

The **for** loop executes a sequence of commands once for each member of a list. It has the following format:

```
for variable<CR>
        in a_list_of_values<CR>
do<CR>
      command 1<CR>
      command 2<CR>
            .
            .
            .
      last command<CR>
done<CR>
```

Figure 7-25: Format of the **for** Loop Construct

For each iteration of the loop, the next member of the list is assigned to the variable given in the for clause. References to that variable may be made anywhere in the commands within the **do** clause.

It is easier to read a shell program if the looping constructs are visually clear. Since the shell ignores spaces at the beginning of lines, each section of commands can be indented as it was in the above format. Also, if you indent each command section, you can easily check to make sure each **do** has a corresponding **done** at the end of the loop.

The variable can be any name you choose. For example, if you call it **var**, then the values given in the list after the keyword **in** will be assigned in turn to **var**; references within the command list to **$var** will make the value available. If the **in** clause is omitted, the values for **var** will be the complete set of arguments given to the command and available in the special parameter **$***. The command list between the keywords **do** and **done** will be executed once for each value.

When the commands have been executed for the last value in the list, the program will execute the next line below **done**. If there is no line, the program will end.

The easiest way to understand a shell programming construct is to try an example. Create a program that will move files to another directory Include the following commands for the purposes shown:

echo	to prompt the user for a path name to the new directory.
read	to assign the path name to the variable **path**
for *variable*	to call the variable **file**; it can be referenced as **$file** in the command sequence.
in *list_of_values*	to supply a list of values. If the **in** clause is omitted, the list of values is assumed to be **$*** (all the arguments entered on the command line).
do *command_sequence*	to provide a command sequence. The construct for this program will be:

do

mv *$file* *$path/$file* <**CR**>
done

The following screen shows the text for the shell program **mv.file**:

```
$ cat mv.file<CR>
echo Please type in the directory path
read path
for file
   in memo1 memo2 memo3
do
   mv $file $path/$file
done
$
```

In this program the values for the variable **file** are already in the program. To change the files each time the program is invoked, assign the values using positional parameters or the **read** command. When positional parameters are used, the **in** keyword is not needed, as the next screen shows:

```
$ cat mv.file<CR>
echo type in the directory path
read path
for file
do
   mv $file $path/$file
done
$
```

You can move several files at once with this command by specifying a list of file names as arguments to the command. (This can be done most easily using the file name expansion mechanism described earlier).

Figure 7-26 summarizes the **mv.file** shell program.

Shell Program Recap **mv.file -** move files to another directory (user defined)	
command	*arguments*
mv.file	*filenames* (interactive)
Description:	Moves files to a new directory.
Remarks:	This program requires file names to be given as arguments. The program prompts for the path to the new directory.

Figure 7-26: Summary of **mv.file** Shell Program

The while *Loop*

Another loop construct, the **while** loop, uses two groups of commands. It will continue executing the sequence of commands in the second group, the **do...done** list, as long as the final command in the first group, the **while** list, returns a status of true (meaning the command can be executed).

The general format of the **while** loop is shown in Figure 7-27.

```
while<CR>
      command 1<CR>
            .
            .
            .
      last command<CR>
do<CR>
      command 1<CR>
            .
            .
            .
      last command<CR>
done<CR>
```

Figure 7-27: Format of the **while** Loop Construct

For example, a program called enter.name uses a **while** loop to enter a list of names into a file. The program consists of the following command lines:

```
$ cat enter.name<CR>
while
   read x
do
   echo $x>>xfile
done
$
```

With some added refinements, the program becomes:

```
$ cat enter.name<CR>
echo Please type in each person's name and then a <CR>
echo Please end the list of names with a <^d>
while read x
do
   echo $x>>xfile
done
echo xfile contains the following names:
cat xfile
$
```

Notice that after the loop is completed, the program executes the commands below the **done**.

You used special characters in the first two **echo** command lines, so you must use quotes to turn off the special meaning. The next screen shows the results of **enter.name**:

```
$ enter.name<CR>
Please type in each person's name and then a <CR>
Please end the list of names with a <^d>
Mary Lou<CR>
Janice<CR>
<^d>
xfile contains the following names:
Mary Lou
Janice
$
```

Notice that after the loop completes, the program prints all the names contained in **xfile**.

The Shell's Garbage Can: /dev/null

The file system has a file called **/dev/null** where you can have the shell deposit any unwanted output.

Try out **/dev/null** by destroying the results of the **who** command. First, type in the **who** command. The response tells you who is on the system. Now, try the **who** command, but redirect the output into **/dev/null**:

```
who > /dev/null<CR>
```

Notice that the system responded with a prompt. The output from the **who** command was placed in **/dev/null** and was effectively discarded.

Conditional Constructs

if...then

The **if** command tells the shell program to execute the **then** sequence of commands only if the final command in the **if** command list is successful. The **if** construct ends with the keyword **fi**.

The general format for the **if** construct is shown in Figure 7-28.

```
if<CR>
      command1<CR>
          .
          .
          .
      last command<CR>
   then<CR>
          command1<CR>
              .
              .
              .
          last command<CR>
fi<CR>
```

Figure 7-28: Format of the **if...then** Conditional Construct

For example, a shell program called search demonstrates the use of the **if...then** construct. **search** uses the **grep** command to search for a word in a file. If **grep** is successful, the program will **echo** that the word is found in the file. Copy the **search** program (shown on the following screen) and try it yourself:

```
$ cat search<CR>
echo Type in the word and the file name.
read word file
if grep $word $file
   then echo $word is in $file
fi
$
```

Notice that the **read** command assigns values to two variables. The first characters you type, up until a space, are assigned to **word**. The rest of the characters, including embedded spaces, are assigned to **file**.

A problem with this program is the unwanted display of output from the **grep** command. If you want to dispose of the system response to the **grep** command in your program, use the file **/dev/null**, changing the **if** command line to the following:

if grep $*word* **$***file* **> /dev/null<CR>**

Now execute your **search** program. It should respond only with the message specified after the **echo** command.

if...then...else

The **if...then** construction can also issue an alternate set of commands with **else**, when the **if** command sequence is false. It has the following general format:

```
if<CR>
      command1<CR>
            .
            .
            .
      last command<CR>
   then<CR>
      command1<CR>
            .
            .
            .
      last command<CR>
   else<CR>
      command1<CR>
            .
            .
            .
      last command<CR>
fi<CR>
```

Figure 7-29: Format of the **if...then...else** Conditional Construct

You can now improve your search command so it will tell you when it cannot find a word, as well as when it can. The following screen shows how your improved program will look:

```
$ cat search<CR>
echo Type in the word and the file name.
read word file
if
   grep $word $file >/dev/null
then
   echo $word is in $file
else
   echo $word is NOT in $file
fi
$
```

Figure 7-30 summarizes your enhanced **search** program.

Shell Program Recap **search -** tells you if a word is in a file (user defined)	
command	*arguments*
search	interactive
Description:	Reports whether a word is in a file.
Remarks:	The command prompts you for the arguments (the word and the file)

Figure 7-30: Summary of the **search** Shell Program

The test *Command for Loops*

The **test** command, which checks to see if certain conditions are true, is a useful command for conditional constructs. If the condition is true, the loop will continue. If the condition is false, the loop will end and the next command will be executed. Some of the useful options for the **test** command are (true if):

test **–r** *file* <**CR**> file exists & is readable

test **–w** *file* <**CR**> file exists & has write permission

test **–x** *file* <**CR**> file exists & is executable

test **–s** *file* <**CR**> file exists & has at least one char.

test *var1* **–eq** *var2* <**CR**> *var1* equals *var2*

test *var1* **–ne** *var2* <**CR**> *var1* does not equal *var2*

You may want to create a shell program to move all the executable files in the current directory to your **bin** directory. You can use the **test** **–x** command to select the executable files. Review the example of the **for** construct that occurs in the **mv.file**

program, shown in the following screen:

```
$ cat mv.file<CR>
echo type in the directory path
read path
for file
do
  mv $file $path/$file
done
$
```

Create a program called **mv.ex** that includes an **if test -x** statement in the **do...done** loop to move executable files only. Your program will be as follows:

```
$ cat mv.ex<CR>
echo type in the directory path
read path
for file
  do
    if test -x $file
       then
         mv $file $path/$file
    fi
  done
$
```

The directory path will be the path from the current directory to the **bin** directory. However, if you use the value for the shell variable **HOME**, you will not need to type in the path each time. **$HOME** gives the path to the login directory. **$HOME/bin** gives

the path to your **bin**.

In the following example, **mv.ex** does not prompt you to type in the directory name, and therefore, does not read the **path** variable:

```
$ cat mv.ex<CR>
for file
  do
    if test -x $file
       then
         mv $file $HOME/bin/$file
    fi
  done
$
```

Test the command, using all the files in the current directory, specified with the * metacharacter as the command argument. The command lines shown in the following example execute the command from the current directory and then changes to **bin** and lists the files in that directory. All executable files should be there.

```
$ mv.ex *<CR>
$ cd; cd bin; ls<CR>
list_of_executable_files
$
```

Figure 7-31 summarizes the format and capabilities of the **mv.ex** shell program.

Shell Program Recap **mv.ex** - move all executable files in the current directory to the **bin** directory	
command	*arguments*
mv.ex	* (all file names)
Description:	Moves all files in the current directory with execute permission to the **bin** directory.
Remarks:	All executable files in the **bin** directory (or any directory shown by the **PATH** variable) can be executed from any directory.

Figure 7-31: Summary of the **mv.ex** Shell Program

case..esac

The **case...esac** construction has a multiple choice format that allows you to choose one of several patterns and then execute a list of commands for that pattern. The pattern statements must begin with the keyword **in**, and a **)** must be placed after the last character of each pattern. The command sequence for each pattern is ended with **;;**. The **case** construction must be ended with **esac** (the letters of the word case reversed).

The general format for the **case** construction shown in Figure 7-32:

```
case word<CR>
in<CR>
    pattern1)<CR>
        command line 1<CR>
            .
            .
            .
        last command line<CR>
    ;;<CR>
    pattern2)<CR>
        command line 1<CR>
            .
            .
            .
        last command line<CR>
    ;;<CR>
    pattern3)<CR>
        command line 1<CR>
            .
            .
            .
        last command line<CR>
    ;;<CR>
    *)<CR>
        command 1<CR>
            .
            .
            .
        last command<CR>
    ;;<CR>
esac<CR>
```

Figure 7-32: The **case...esac** Conditional Construct

The **case** construction tries to match the *word* following the word **case** with the *pattern* in the first pattern section. If there is a match, the program executes the command lines after the first pattern and up to the corresponding ;;.

If the first pattern is not matched, the program proceeds to the second pattern. Once a pattern is matched, the program does not try to match any more of the patterns, but goes to the command following **esac**.

The * used as a pattern matches any *word*, and so allows you to give a set of commands to be executed if no other pattern matches. To do this, it must be placed as the last possible pattern in the **case** construct, so that the other patterns are checked first. This provides a useful way to detect erroneous or unexpected

input.

The patterns that can be specified in the *pattern* part of each section may use the metacharacters *****, **?**, and **[]** as described earlier in this chapter for the shell's file name expansion capability. This provides useful flexibility.

The **set.term** program contains a good example of the **case...esac** construction. This program sets the shell variable TERM according to the type of terminal you are using. It uses the following command line:

TERM = *terminal_name* < **CR** >

(For an explanation of the commands used, see the **vi** tutorial in Chapter 6.) In the following example, the terminal is a Teletype 4420, Teletype 5410, or Teletype 5420.

set.term first checks to see whether the value of **term** is 4420. If it is, the program makes T4 the value of **TERM**, and terminates. If it the value of **term** is not 4420, the program checks for other values: 5410 and 5420. It executes the commands under the first pattern that it finds, and then goes to the first command after the **esac** command.

The pattern ***** , meaning everything else, is included at the end of the terminal patterns. It will warn that you do not have a pattern for the terminal specified and will allow you to exit the **case** construct:

```
$ cat set.term <CR>
echo If you have a TTY 4420 type in 4420
echo If you have a TTY 5410 type in 5410
echo If you have a TTY 5420 type in 5420
read term
case $term
        in
            4420)
                TERM=T4
            ;;
            5410)
                TERM=T5
            ;;
            5420)
                TERM=T7
            ;;
            *)
            echo not a correct terminal type
            ;;
esac
export TERM
echo end of program
$
```

Notice the use of the **export** command. You use **export** to make a variable available within your environment and to other shell procedures. What would happen if you placed the * pattern first? The **set.term** program would never assign a value to TERM, since it would always match the first pattern *, which means everything.

Figure 7-33 summarizes the format and capabilities of the **set.term** shell program.

Shell Program Recap **set.term** - assign a value to TERM (user defined)	
command	*arguments*
set.term	interactive
Description:	Assigns a value to the shell variable TERM and then exports that value to other shell procedures.
Remarks:	This command asks for a specific terminal code to be used as a pattern for the **case** construction.

Figure 7-33: Summary of the **set.term** Shell Program

Unconditional Control Statements: the break and continue Commands

The **break** command unconditionally stops the execution of any loop in which it is encountered, and goes to the next command after the **done, fi**, or **esac** statement. If there are no commands after that statement, the program ends.

In the example for **set.term**, you could have used the **break** command instead of **echo** to leave the program, as the next example shows:

```
$ cat set.term <CR>
echo If you have a TTY 4420 type in 4420
echo If you have a TTY 5410 type in 5410
echo If you have a TTY 5420 type in 5420
read term
case $term
        in
            4420)
            TERM=T4
        ;;
        5410)
            TERM=T5
        ;;
        5420)
            TERM=T7
        ;;
        *)
            break
        ;;
esac
export TERM
echo end of program
$
```

The **continue** command causes the program to go immediately to the next iteration of a **do** or **for** loop without executing the remaining commands in the loop.

Debugging Programs

At times you may need to debug a program to find and correct errors. There are two options to the **sh** command (listed below) that can help you debug a program:

sh **–v** *shellprogramname* prints the shell input lines as they are read by the system

sh **−x** *shellprogramname* prints commands and their arguments as they are executed

To try out these two options, create a shell program that has an error in it. For example, create a file called **bug** that contains the following list of commands:

```
$ cat bug<CR>
today=`date`
echo enter person
read person
mail $1
$person
When you log off come into my office please.
$today.
MLH
$
```

Notice that **today** equals the output of the **date** command, which must be enclosed in grave accents for command substitution to occur.

The mail message sent to Tom ($1) at login **tommy** ($2) should read as the following screen shows:

```
$ mail<CR>
From mlh  Thu  Apr 10  11:36  CST  1984
Tom
When you log off come into my office please.
Thu  Apr 10  11:36:32  CST  1986
MLH
?
```

If you try to execute **bug**, you will have to press the BREAK or DELETE key to end the program.

To debug this program, try executing **bug** using **sh -v**. This will print the lines of the file as they are read by the system, as shown below:

```
$ sh -v bug tom<CR>
today=`date`
echo enter person
enter person
read person
tommy
mail $1
```

Notice that the output stops on the **mail** command, since there is a problem with **mail**. You must use the here document to redirect input into **mail**.

Before you fix the **bug** program, try executing it with **sh -x**, which prints the commands and their arguments as they are read by the system:

```
$ sh -x bug tom tommy<CR>
+date
today=Thu  Apr 10  11:07:23  CST  1986
+ echo enter person
enter person
+ read person
tommy
+ mail tom
$
```

Once again, the program stops at the **mail** command. Notice that the substitutions for the variables have been made and are displayed.

The corrected **bug** program is as follows:

```
$ cat bug<CR>
today=`date`
echo enter person
read person
mail $1 <<!
$person
When you log off come into my office please.
$today
MLH
!
$
```

The **tee** command is a helpful command for debugging pipelines. While simply passing its standard input to its standard output, it also saves a copy of its input into the file whose name is given as an argument.

The general format of the **tee** command is:

command1 ¦ **tee** *saverfile* ¦ *command2*<**CR**

saverfile is the file that saves the output of *command1* for you to study.

For example, say you want to check on the output of the **grep** command in the following command line:

who ¦ grep $1 ¦ cut –c1-9<CR>

You can use **tee** to copy the output of **grep** into a file called **check**, without disturbing the rest of the pipeline.

who ¦ grep $1 ¦ tee check ¦ cut –c1-9<CR>

The file **check** contains a copy of the **grep** output, as shown in the following screen:

```
$ who ¦ grep mlhmo ¦ tee check ¦ cut –c1-9<CR>
mlhmo
$ cat check<CR>
mlhmo    tty61    Apr 10    11:30
$
```

For further information about shell programming, including features such as command return codes, refer to the manual *Shell Commands and Programming*.

Modifying Your Login Environment

The UNIX system lets you modify your login environment in several ways. One modification that users commonly want to make is to change the default values of the erase (#) and line kill (@) characters.

When you log in, the shell first examines a file in your login directory named **.profile** (pronounced "dot profile"). This file contains commands that control your shell environment.

Because the **.profile** is a file, it can be edited and changed to suit your needs. On some systems you can edit this file yourself, while on others, the system administrator does this for you. To see whether you have a **.profile** in your home directory, type:

ls -al $HOME

If you can edit the file yourself, you may want to be cautious the first few times. Before making any changes to your **.profile**, make a copy of it in another file called **safe.profile**. Type:

cp .profile safe.profile<CR>

You can add commands to your **.profile** just as you add commands to any other shell program. You can also set some terminal options with the **stty** command, and set some shell variables.

Adding Commands to Your .profile

Practice adding commands to your **.profile**. Edit the file and add the following **echo** command to the last line of the file:

echo Good Morning! I am ready to work for you.

Write and quit the editor.

Whenever you make changes to your **.profile** and you want to initiate them in the current work session, you may cause the commands in **.profile** to be executed directly using the **.** (dot) shell command. The shell will reinitialize your environment by reading executing the commands in your **.profile**. Try this now. Type:

. .profile<CR>

The system should respond with the following:

```
Good Morning! I am ready to work for you.
$
```

Setting Terminal Options

The **stty** command can make your shell environment more convenient. There are three options you can use with **stty**: **-tabs**, **erase <^h>**, and **echoe**.

stty –tabs	This option preserves tabs when you are printing. It expands the tab setting to eight spaces, which is the default. The number of spaces for each tab can be changed. (See **stty**(1) in the *User's Reference Manual* for details.)
stty erase <^h>	This option allows you to use the erase key on your keyboard to erase a letter, instead of the default character #. Usually the BACKSPACE key is the erase key.
stty echoe	If you have a terminal with a screen, this option erases characters from the screen as you erase them with the BACKSPACE key.

If you want to use these options for the **stty** command, you can create those command lines in your **.profile** just as you would create them in a shell program. If you use the **tail** command, which displays the last few lines of a file, you can see the results of adding those four command lines to your **.profile**:

```
$ tail -4 .profile<CR>
echo Good Morning! I am ready to work for you
stty -tabs
stty erase <^h>
stty echoe
$
```

Figure 7-34 summarizes the format and capabilities of the **tail** command.

Command Recap **tail** - display the last portion of a file		
command	*options*	*arguments*
tail	*-n*	*filename*
Description:	Displays the last lines of a file.	
Options:	Use *-n* to specify the number of lines *n* (default is ten lines). You can specify a number of blocks (**-*n*b**) or characters (**-*n*c**) instead of lines.	

Figure 7-34: Summary of the **tail** Command

Creating an rje Directory

We have often talked about sharing useful programs with other users in this chapter. Similarly, these users may have programs or other files that they want to share with you. So that these users can send you the files easily, you should create an **rje** (remote job entry) directory:

```
mkdir rje
chmod go+w rje
```

Notice that you have to change the permissions of the directory using **chmod**. When you have an **rje** directory with the correct permissions, other users can send you files using the **uucp** command. See the **uucp**(1) manual page in the *User's Reference Manual* for details.

Using Shell Variables

Several of the variables reserved by the shell are used in your **.profile**. You can display the current value for any shell variable by entering the following command:

echo $*variable_name*

Four of the most basic of these variables are discussed next.

HOME

This variable gives the path name of your login directory. Use the **cd** command to go to your login directory and type:

pwd<CR>

What was the system response? Now type:

echo $HOME<CR>

Was the system response the same as the response to **pwd**?

$HOME is the default argument for the **cd** command. If you do not specify a directory, **cd** will move you to **$HOME**.

PATH

This variable gives the search path for finding and executing commands. To see the current values for your **PATH** variable type:

echo $PATH<CR>

The system will respond with your current PATH value.

```
$ echo $PATH<CR>
:/mylogin/bin:/bin:/usr/bin:/usr/lib
$
```

The colon (**:**) is a delimiter between path names in the string assigned to the $PATH variable. When nothing is specified before a **:** , then the current directory is understood. Notice how, in the last example, the system looks for commands in the current directory first, then in **/mylogin/bin/**, then in **/bin**, then in **/usr/bin**, and finally in **/usr/lib**.

If you are working on a project with several other people, you may want to set up a group **bin**, a directory of special shell programs used only by your project members. The path might be named **/project1/bin**. Edit your **.profile**, and add **:/project1/bin** to the end of your **PATH**, as in the next example.

PATH = ":/mylogin/bin:/bin:/usr/lib:/project1/bin" < CR >

TERM

This variable tells the shell what kind of terminal you are using. To put assign a value to it, you must execute the following three commands in this order:

TERM = *terminal_name* **< CR >**
export TERM < CR >
tput init

The first two lines, together, are necessary to tell the computer what type of terminal you are using. (For an explanation of exporting variables, see *Shell Commands and Programming*.) The last line, containing the **tput** command, tells the terminal that the computer is expecting to communicate with the type of terminal specified in the TERM variable. Therefore this command must always be entered after the variable has been exported.

If you do not want to specify the **TERM** variable each time you log in, add these three command lines to your **.profile**; they will be executed automatically whenever you log in. To determine what terminal name to assign to the TERM variable, see the instructions in Appendix F, "Setting Up the Terminal." This appendix also contains details about the **tput** command.

If you log in on more than one type of terminal, it would also be useful to have your **set.term** command in your **.profile**.

PS1

This variable sets the primary shell prompt string (the default is the $ sign). You can change your prompt by changing the **PS1** variable in your **.profile**.

Try the following example. Note that to use a multiword prompt, you must enclose the phrase in quotes. Type the following variable assignment in your **.profile**.

PS1 = "Your command is my wish < CR > "

Now execute your **.profile** (with the **.** command) and watch for your new prompt sign.

```
$ . .profile<CR>
Your command is my wish
```

The mundane $ sign is gone forever, or at least until you delete the **PS1** variable from your **.profile**.

Shell Programming Exercises

2-1. Create a shell program called **time** from the following command line:

banner `date ¦ cut –c12-19` <CR>

2-2. Write a shell program that will give only the date in a banner display. Be careful not to give your program the same name as a UNIX system command.

2-3. Write a shell program that will send a note to several people on your system.

2-4. Redirect the **date** command without the time into a file.

2-5. Echo the phrase Dear colleague in the same file that contains the date command, without erasing the date.

2-6. Using the above exercises, write a shell program that will send a memo to the same people on your system mentioned in Exercise 2-3. Include in your memo:

The current date and the words Dear colleague at the top of the memo

The body of the memo (stored in an existing file)

The closing statement

2-7. How can you **read** variables into the **mv.file** program?

2-8. Use a **for** loop to move a list of files in the current directory to another directory. How can you move all your files to another directory?

2-9. How can you change the program **search**, so that it searches through several files?

Hint:

for file
in $*

2-10. Set the **stty** options for your environment.

2-11. Change your prompt to the word Hello.

2-12. Check the settings of the variables **$HOME**, **$TERM**, and **$PATH** in your environment.

Answers To Exercises

Command Language Exercises

1-1. The * at the beginning of a file name refers to all files that end in that file name, including that file name.

```
$ ls *t<CR>
cat
123t
new.t
t
$
```

1-2. The command **cat [0-9]*** will produce the following output:

```
1memo
100data
9
05name
```

The command **echo *** will produce a list of all the files in the current directory.

1-3. You can place **?** in any position in a file name.

1-4. The command **ls [0-9]*** will list only those files that start with a number.

The command **ls [a-m]*** will list only those files that start with the letters "a" through "m".

1-5. If you placed the sequential command line in the background mode, the immediate system response was the PID number for the job.

No, the **&** (ampersand) must be placed at the end of the command line.

1-6. The command line would be:

cd; pwd > junk; ls >> junk; ed trial<CR>

1-7. Change the **-c** option of the command line to read:

banner `date | cut -c1-10`<CR>

Shell Programming Exercises

2-1.

```
$ cat time<CR>
banner `date | cut -c12-19`
$
$ chmod u+x time<CR>
$ time<CR>
(banner display of the time 10:26)
$
```

2-2.

```
$ cat mydate<CR>
banner `date | cut -c1-10`
$
```

2-3.

```
$ cat tofriends<CR>
echo Type in the name of the file containing the note.
read note
mail janice marylou bryan < $note
$
```

Or, if you used parameters for the logins, instead of the logins themselves, your program may have looked like this:

```
$ cat tofriends<CR>
echo Type in the name of the file containing the note.
read note
mail $* < $note
$
```

2-4. **date ¦ cut –c1-10 > file1<CR>**

2-5. **echo Dear colleague >> file1<CR>**

2-6.

```
$ cat send.memo<CR>
date ¦ cut -c1-10 > memo1
echo Dear colleague >> memo1
cat memo >> memo1
echo A memo from M. L. Kelly >> memo1
mail janice marylou bryan < memo1
$
```

2-7.

```
$ cat mv.file<CR>
echo type in the directory path
read path
echo type in file names, end with <^d>
  while
  read file
    do
      mv $file $path/$file
    done
echo all done
$
```

2-8.

```
$ cat mv.file<CR>
echo Please type in directory path
read path
for file in $*
  do
    mv $file $path/$file
  done
$
```

The command line for moving all files in the current directory is:

$ mv.file * <CR>

2-9. See hint given with exercise 2-9.

```
$ cat search<CR>
for file
  in $*
  do
    if grep $word $file >/dev/null
    then echo $word is in $file
    else echo $word is NOT in $file
    fi
  done
$
```

2-10. Add the following lines to your **.profile**.

```
stty -tabs<CR>
stty erase <^h><CR>
stty echoe<CR>
```

2-11. Add the following command lines to your **.profile**

```
PS1=Hello<CR>
export PS1
```

2-12. To check the values of these variables in your home environment:

```
$ echo $HOME<CR>
$ echo $TERM<CR>
$ echo $PATH<CR>
```

8. ELECTRONIC COMMUNICATIONS TUTORIALS

Chapter 8: Communication Tutorial

Introduction 8-1

Exchanging Messages 8-2

mail 8-3
- Sending Messages 8-3
  - Undeliverable Mail 8-4
  - Sending Mail to One Person 8-6
  - Sending Mail to Several People Simultaneously 8-7
- Sending Mail to Remote Systems: the **uname** and **uuname** Commands 8-8
- Managing Incoming Mail 8-12

mailx 8-16

mailx Overview 8-17

Command Line Options 8-19

How to Send Messages: the Tilde Escapes 8-20
- Editing the Message 8-22
- Incorporating Existing Text into Your Message 8-24
  - Reading a File into a Message 8-25
  - Incorporating a Message from Your Mailbox into a Reply 8-26
- Changing Parts of the Message Header 8-27
- Adding Your Signature 8-28
- Keeping a Record of Messages You Send 8-29
- Exiting from **mailx** 8-31
- Summary 8-31

How to Manage Incoming Mail 8-32
The *msglist* Argument 8-32
Commands for Reading and Deleting Mail 8-33
Reading Mail 8-33
Scanning Your Mailbox 8-35
Switching to Other Mail Files 8-36
Deleting Mail 8-37
Commands for Saving Mail 8-38
Commands for Replying to Mail 8-39
Commands for Getting Out of **mailx** 8-40
mailx Command Summary 8-41

The **.mailrc** File 8-42

Sending and Receiving Files 8-47
Sending Small Files: the **mail** Command 8-47
Sending Large Files 8-48
Getting Ready: Do You Have Permission? 8-49
The **uucp** Command 8-51
Command Line Syntax 8-52
Sample Usage of Options with the **uucp** Command 8-54
How the **uucp** Command Works 8-56
The **uuto** Command 8-58
Sending a File: the **202m** Option and **uustat** Command 8-58
Receiving Files Sent with **uuto**: the **uupick** Command 8-63

Networking 8-67
Connecting a Remote Terminal: the **ct** Command 8-67
Command Line Format 8-68
Sample Command Usage 8-68

Calling Another UNIX System: the **cu** Command 8-70
Command Line Format 8-71
Sample Command Usage 8-75
Executing Commands on a Remote System: the **uux** Command 8-76
Command Line Format 8-77
Sample Command Usage 8-77

Introduction

The UNIX system offers a choice of commands that enable you to communicate with other UNIX system users. Specifically, they allow you to: send and receive messages from other users (on either your system or another UNIX system); exchange files; and form networks with other UNIX systems. Through networking, a user on one system can exchange messages and files between computers, and execute commands on remote computers.

To help you take advantage of these capabilities, this chapter will teach you how to use the following commands.

For exchanging messages:	**mail**, **mailx**, **uname**, and **uuname**
For exchanging files:	**uucp**, **uuto**, **uupick**, and **uustat**
For networking:	**ct**, **cu**, and **uux**

Exchanging Messages

To send messages you can use either the **mail** or **mailx** command. These commands deliver your message to a file belonging to the recipient. When the recipient logs in (or while already logged in), he or she receives a message that says `you have mail`. The recipient can use either the **mail** or **mailx** command to read your message and reply at his or her leisure.

The main difference between **mail** and **mailx** is that only **mailx** offers the following features:

- a choice of text editors (**ed** or **vi**) for handling incoming and outgoing messages
- a list of waiting messages that allows the user to decide which messages to handle and in what order
- several options for saving files
- commands for replying to messages and sending copies (of both incoming and outgoing messages) to other users

You can also use **mail** or **mailx** to send short files containing memos, reports, and so on. However, if you want to send someone a file that is over a page long, use one of the commands designed for transferring files: **uuto** or **uucp**. (See "Sending Large Files" later in this chapter for descriptions of these commands.)

mail

This section presents the **mail** command. It discusses the basics of sending mail to one or more people simultaneously, whether they are working on the local system (the same system as you) or on a remote system. It also covers receiving and handling incoming mail.

Sending Messages

The basic command line format for sending mail is

mail *login* <**CR**>

where *login* is the recipient's login name on a UNIX system. This login name can be either of the following:

- a login name if the recipient is on your system (for example, **bob**)
- a system name and login name if the recipient is on another UNIX system that can communicate with yours (for example, **sys2!bob**)

For the moment, assume that the recipient is on the local system. (We will deal with sending mail to users on remote systems later.) Type the **mail** command at the system prompt, press the RETURN key, and start typing the text of your message on the next line. There is no limit to the length of your message. When you have finished typing it, send the message by typing a period (.) or a <^d> (control-d) at the beginning of a new line.

The following example shows how this procedure will appear on your screen.

```
$ mail phyllis<CR>
My meeting with Smith's<CR>
group tomorrow has been moved<CR>
up to 3:00 so I won't be able to<CR>
see you then.  Could we meet<CR>
in the morning instead?<CR>
.<CR>
$
```

The prompt on the last line means that your message has been queued (placed in a waiting line of messages) and will be sent.

Undeliverable Mail

If you make an error when typing the recipient's login, the **mail** command will not be able to deliver your mail. Instead, it will print two messages telling you that it has failed and that it is returning your mail. Then it will return your mail in a message that includes the system name and login name of both the sender and intended recipient, and an error message stating the reason for the failure.

For example, say you (owner of the login **kol**) want to send a message to a user with the login **chris** on a system called **marmaduk**. Your message says `The meeting has been changed to 2:00.` Failing to notice that you have incorrectly typed the login as **cris**, you try to send your message.

```
$ mail cris<CR>
The meeting has been changed to 2:00.
.<CR>
mail: Can't send to cris
mail: Return to kol
you have mail in /usr/mail/kol
$
```

The mail that is waiting for you in **/usr/mail** will be useful if you do not know why the **mail** command has failed, or if you want to retrieve your mail so that you can resend it without typing it in again. It contains the following:

```
$ mail<CR>
From kol Sat Jan 18 17:33 EST 1986
>From kol Sat Jan 18 17:33 EST 1986 forwarded by kol
***** UNDELIVERABLE MAIL sent to cris, being returned by
                                    marmaduk!kol *****
mail: ERROR # 8 'Invalid recipient' encountered on system
                                    marmaduk

The meeting has been changed to 2:00.

?
```

To learn how to display and handle this message see "Managing Incoming Mail" later in this chapter.

Sending Mail to One Person

The following screen shows a typical message.

```
$ mail tommy<CR>
Tom,<CR>
There's a meeting of the review committee<CR>
at 3:00 this afternoon.  D.F. wants your<CR>
comments and an idea of how long you think<CR>
the project will take to complete.<CR>
B.K.<CR>
.<CR>
$
```

When Tom logs in at his terminal (or while he is already logged in), he receives a message that tells him he has mail waiting:

```
$ you have mail
```

To find out how he can read his mail, see the section "Managing Incoming Mail" in this chapter.

You can practice using the **mail** command by sending mail to yourself. Type in the **mail** command and your login ID, and then write a short message to yourself. When you type the final period or <^d>, the mail will be sent to a file named after your login ID in the **/usr/mail** directory, and you will receive a notice that you have mail.

Sending mail to yourself can also serve as a handy reminder system. For example, suppose you (login ID **bob**) want to call someone the next morning. Send yourself a reminder in a mail message.

```
$ mail bob<CR>
Call Accounting and find out<CR>
why they haven't returned my 1985 figures!<CR>
.<CR>
$
```

When you log in the next day, a notice will appear on your screen informing you that you have mail waiting to be read.

Sending Mail to Several People Simultaneously

You can send a message to a number of people by including their login names on the **mail** command line. For example:

```
$ mail tommy jane wombat dave<CR>
Diamond cutters,<CR>
The game is on for tonight at diamond three.<CR>
Don't forget your gloves!<CR>
Your Manager<CR>
.<CR>
$
```

Figure 8-1 summarizes the syntax and capabilities of the **mail** command.

Command Recap		
mail - sends a message to another user's login		
command	*options*	*arguments*
mail	none	*[system_name!]login*
Description:	Typing **mail** followed by one or more login names, sends the message typed on the lines following the command line to the specified login(s).	
Remarks:	Typing a period or a <^d> (followed by the RETURN key) at the beginning of a new line sends the message.	

Figure 8-1: Summary of Sending Messages with the **mail** Command

Sending Mail to Remote Systems: the uname and uuname Commands

Until now we have assumed that you are sending messages to users on the local UNIX system. However, your company may have three separate computer systems, each in a different part of a building, or you may have offices in several locations, each with its own system.

You can send mail to users on other systems simply by adding the name of the recipient's system before the login ID on the command line.

mail sys2!bob<CR>

Notice that the system name and the recipient's login ID are

separated by an exclamation point.

Before you can run this command, however, you need three pieces of information:

- the name of the remote system
- whether or not your system and the remote system communicate
- the recipient's login name

The **uname** and **uuname** commands allow you to find this information.

If you can, get the name of the remote system and the recipient's login name from the recipient. If the recipient does not know the system name, have him or her issue the following command on the remote system:

uname -n <CR>

The command will respond with the name of the system. For example:

```
$ uname -n <CR>
dumbo
$
```

Once you know the remote system name, the **uuname** command can help you verify that your system can communicate with the remote system. At the prompt, type:

uuname <CR>

This generates a list of remote systems with which your system can communicate. If the recipient's system is on that list, you can send messages to it by **mail**.

You can simplify this step by using the **grep** command to search through the **uuname** output. At the prompt, type:

uuname ¦ **grep** *system* <CR>

(Here *system* is the recipient's system name.) If **grep** finds the specified system name, it prints it on the screen. For example:

```
$ uuname | grep dumbo<CR>
dumbo
$
```

This means that **dumbo** can communicate with your system. If **dumbo** does not communicate with your system, **uuname** returns a prompt.

```
$ uuname | grep dumbo<CR>
$
```

To summarize our discussion of **uname** and **uuname**, consider an example. Suppose you want to send a message to login **sarah** on the remote system dumbo. Verify that dumbo can communicate with your system and send your message. The following screen shows both steps.

```
$ uuname | grep dumbo<CR>
dumbo
$ mail dumbo!sarah<CR>
Sarah,<CR>
The final counts for the writing seminar<CR>
are as follows:<CR>
<CR>
Our department - 18<CR>
Your department - 20<CR>
<CR>
Tom<CR>
.<CR>
$
```

Figures 8-2 and 8-3 summarize the syntax and capabilities of the **uname** and **uuname** commands, respectively.

Command Recap **uname** - displays the system name		
command	*options*	*arguments*
uname	**-n** and others*	none
Description:	**uname -n** displays the name of the system on which your login resides.	

Figure 8-2: Summary of the **uname** Command

* See the **uname**(1) manual page in the *User's Reference Manual* for all available options and an explanation of their capabilities.

Command Recap **uuname** - displays a list of networked systems		
command	*options*	*arguments*
uuname	none	none
Description:	**uuname** displays a list of remote systems that can communicate with your system.	

Figure 8-3: Summary of the **uuname** Command

Managing Incoming Mail

As stated earlier, the **mail** command also allows you to display messages sent to you by other users on your screen so you can read them. If you are logged in when someone sends you mail, the following message is printed on your screen:

```
you have mail
```

This means that one or more messages are being held for you in a file called **/usr/mail/***your_login*, usually referred to as your mailbox. To display these messages on your screen, type the **mail** command without any arguments:

mail <CR>

The messages will be displayed one at a time, beginning with the one most recently received. A typical **mail** message display looks like this:

```
$ mail
From tommy Wed May 21 15:33 CST 1986
Bob,
Looks like the meeting has been cancelled.
Do you still want the material for the technical review?
Tom

?
```

The first line, called the header, provides information about the message: the login name of the sender and the date and time the message was sent. The lines after the header (up to the line containing the ?) comprise the text of the message.

If a long message is being displayed on your terminal screen, you may not be able to read it all at once. You can interrupt the printing by typing <^s> (control-s). This will freeze the screen, giving you a chance to read. When you are ready to continue, type <^q> and the printing will resume.

After displaying each message, the **mail** command prints a ? prompt and waits for a response. You have many options, for example, you can leave the current message in your mailbox while you read the next message; you can delete the current message; or you can save the current message for future reference. For a list of **mail**'s available options, type a **?** in response to **mail**'s ? prompt.

To display the next message without deleting the current message, press the RETURN key after the question mark.

?<CR>

The current message remains in your mailbox and the next message is displayed. If you have read all the messages in your mailbox, a prompt appears.

To delete a message, type a **d** after the question mark:

? d<CR>

The message is deleted from your mailbox. If there is another message waiting, it is then displayed. If not, a prompt appears as a signal that you have finished reading your messages.

To save a message for later reference, type an **s** after the question mark:

? s<CR>

This saves the message, by default, in a file called **mbox** in your home directory. To save the message in another file, type the name of that file after the **s** command.

For example, to save a message in a file called **mailsave** (in your current directory), enter the response shown after the question mark:

? **s mailsave<CR>**

If **mailsave** is an existing file, the **mail** command appends the message to it. If there is no file by that name, the **mail** command creates one and stores your message in it. You can later verify the existence of the new file by using the **ls** command. (**ls** lists the contents of your current directory.)

You can also save the message in a file in a different directory by specifying a path name. For example:

? **s project1/memo<CR>**

This is a relative path name that identifies a file called **memo** (where your message will be saved) in a subdirectory (**project1**) of your current directory. You can use either relative or full path names when saving mail messages. (For instructions on using path names, see Chapter 3.)

To quit reading messages, enter the response shown after the question mark:

? **q<CR>**

Any messages that you have not read are kept in your mailbox until the next time you use the **mail** command.

To stop the printing of a message entirely, press the BREAK key. The **mail** command will stop the display, print a ? prompt, and wait for a response from you.

Figure 8-4 summarizes the syntax and capabilities of the **mail** command for reading messages.

Command Recap **mail** - reads messages sent to your login		
command	*options*	*arguments*
mail	available*	none
Description:	When issued without options, the **mail** command displays any messages waiting in your mailbox (the system file /**usr**/**mail**/*your_login*).	
Remarks:	A question mark (**?**) at the end of a message means that a response is expected. A full list of possible responses is given in the *User's Reference Manual*.	

Figure 8-4: Summary of Reading Messages with the **mail** Command

* See the **mail**(1) manual page in the *User's Reference Manual* for all available options and an explanation of their capabilities.

mailx

This section introduces the **mailx** facility. It explains how to set up your **mailx** environment, send messages with the **mailx** command, and handle messages that have been sent to you. The material is presented in four parts:

- **mailx** Overview
- Sending Messages
- Managing Incoming Mail
- The **.mailrc** File

mailx Overview

The **mailx** command is an enhanced version of the **mail** command. There are many options to **mailx** that are not available in **mail** for sending and reading mail. For example, you can define an alias for a single login or for a group. This allows you to send **mail** to an individual using a name or word other than their login ID, and to send **mail** to a whole group of people using a single name or word. When you use **mailx** to read incoming mail you can save it in various files, edit it, forward it to someone else, respond to the person who originated the message, and so forth. By using **mailx** environment variables you can develop an environment to suit your individual tastes.

If you type the **mailx** command with one or more logins as arguments, **mailx** decides you are sending mail to the named users, prompts you for a summary of the subject, and then waits for you to type in your message or issue a command. The section "How to Send Messages" describes features that are available to you for editing, incorporating other files, adding names to copy lists, and more.

If you enter the **mailx** command with no arguments, **mailx** checks incoming mail for you in a file named /**usr**/**mail**/*your_login*. If there is mail for you in that file, you are shown a list of the items and given the opportunity to read, store, remove or transfer each one to another file. The section entitled "How to Manage Incoming Mail" provides some examples and describes the options available.

If you choose to customize **mailx**, you should create a start-up file in your home directory called **.mailrc**. The section on "The **.mailrc** File" describes variables you can include in your start-up file.

mailx has two modes of functioning: input mode and command mode. You must be in input mode to create and send messages. Command mode is used to read incoming mail. You can use any of the following methods to control the way **mailx** works for you:

- by entering options on the command line. (See the **mailx**(1) manual page in the *User's Reference Manual*.)

- by issuing commands when you are in input mode, for example, creating a message to send. These commands are always preceded by a ~ (tilde) and are referred to as tilde escapes. (See the **mailx**(1) manual page in the *User's Reference Manual*.)
- by issuing commands when you are in command mode, for example, reading incoming mail.
- by storing commands and environment variables in a start-up file in your home directory called **$HOME/.mailrc**.

Tilde escapes are discussed in "Sending Messages," command mode commands in "Managing Incoming Mail," and the **.mailrc** file in "The **.mailrc** File."

Command Line Options

In this section, we will look at command line options.

The syntax for the **mailx** command is:

mailx [*options*] [*name...*]

The *options* are flags that control the action of the command, and *name...* represents the intended recipients.

Anything on the command line other than an option preceded by a hyphen is read by **mailx** as a *name*; that is, the login or alias of a person to whom you are sending a message.

Two of the command line options deserve special mention:

- **-f** [*filename*]: Allows you to read messages from *filename* instead of your mailbox.

  Because **mailx** lets you store messages in any file you name, you need the **-f** option to review these stored options. The default storage file is **$HOME/mbox**, so the command:

  mailx -f

  is used to review messages stored there.

- **-n**: Do not initialize from the system default **mailx.rc** file.

  If you have your own **.mailrc** file (see "The **.mailrc** File") **mailx** will not look through the system default file for specifications when you use the **-n** option, but will go directly to your **.mailrc** file. This results in faster initialization; substantially faster when the system is busy.

How to Send Messages: the Tilde Escapes

To send a message to another UNIX system user, enter the following command:

$ **mailx daves<CR>**

The login name specified belongs to the person who is to receive the message. The system puts you into input mode and prompts you for the subject of the message. (You may have to wait a few seconds for the `Subject:` prompt if the system is very busy.) This is the simplest way to run the **mailx** command; it differs very little from the way you run the **mail** command.

The following examples show how you can edit messages you are sending, incorporate existing text into your messages, change the header information, and perform other tasks that take advantage of the **mailx** command's capabilities. Each example is followed by an explanation of the key points illustrated in the example.

```
$ mailx daves<CR>
Subject:
```

Whether to include a subject or not is optional. If you elect not to, press the RETURN key. The cursor moves to the next line and the program waits for you to enter the text of the message.

```
$ mailx daves<CR>
Subject: meeting<CR>
We're having a meeting for novice mailx users in<CR>
the auditorium at 9:00 tomorrow.<CR>
Would you be willing to give a demonstration?<CR>
Bob<CR>
~. <CR>
cc:<CR>
$
```

There are two important things to notice about the above example:

- You break up the lines of your message by pressing the RETURN key at the end of each line. This makes it easier for the recipient to read the message, and prevents you from overflowing the line buffer.
- You end the text and send the message by entering a tilde and a period together (~.) at the beginning of a line. The system responds with an end-of-text notice (EOT) and a prompt.

There are several commands available to you when you are in input mode (as we were in the example). Each of them consists of a tilde (~), followed by an alphabetic character, entered at the beginning of a line. Together they are known as tilde escapes. (See the **mailx**(1) manual page in the *User's Reference Manual*.) Most of them are used in the examples in this section.

You can include the subject of your message on the command line by using the **-s** option. For example, the command line:

```
$ mailx -s "meeting" daves<CR>
```

is equivalent to:

```
$ mailx daves<CR>
Subject: meeting<CR>
```

The subject line will look the same to the recipient of the message. Notice that when putting the subject on the command line, you must enclose a subject that has more than one word in quotation marks.

Editing the Message

When you are in the input mode of **mailx**, you can invoke an editor by entering the **~e** (tilde e) escape at the beginning of a line. The following example shows how to use tilde:

```
$ mailx daves<CR>
Subject: Testing my tilde<CR>
When entering the text of a message<CR>
that has somehow gotten grabled<CR>
you may invoke your favorite editor<CR>
by means of a ~e (tilde e).
                    .
                    .
                    .
```

Notice that you have misspelled a word in your message. To correct the error, use **~e** to invoke the editor, in this case the default editor, **ed**.

```
                                        .
                                        .
                                        .
~e<CR>
12
/grabled/p
that has somehow gotten grabled
s/gra/gar/p
that has somehow gotten garbled
w
132
q
(continue)
What more can I tell you?
                                        .
                                        .
                                        .
```

In this example the **ed** editor was used. Your **.profile** or a **.mailrc** file controls which editor will be invoked when you issue a **~e** escape command. The **~v** (tilde v) escape invokes an alternate editor (most commonly, **vi**).

When you exited from **ed** (by typing **q**), the **mailx** command returned you to input mode and prompted you to continue your message. At this point you may want to preview your corrected message by entering a **~p** (tilde p) escape. The **~p** escape prints out the entire message up to the point where the **~p** was entered. Thus, at any time during text entry, you can review the current contents of your message.

```
                                        .
                                        .
~p
Message contains:
To: daves
Subject: Testing my tilde

When entering the text of a message
that has somehow gotten garbled
you may invoke your favorite editor
by means of a tilde e (~e).
What more can I tell you?
(continue)
~.
EOT
$
```

Incorporating Existing Text into Your Message

mailx provides four ways to incorporate material from another source into the message you are creating. You can:

- read a file into your message
- read a message you have received into a reply
- incorporate the value of a named environment variable into a message
- execute a shell command and incorporate the output of the command into a message

The following examples show the first two of these functions. These are the most commonly used of these four functions. For information about the other two, see the **mailx**(1) manual page of the *User's Reference Manual*.

Reading a File into a Message

```
$ mailx daves<CR>
Subject: Work Schedule<CR>
As you can see from the following<CR>
~r letters/file1
"letters/file1"    10/725
we have our work cut out for us.
Please give me your thoughts on this.
- Bob
~
.
EOT
$
```

As the example shows, the **~r** (tilde r) escape is followed by the name of the file you want to include. The system displays the file name and the number of lines and characters it contains. You are still in input mode and can continue with the rest of the message. When the recipient gets the message, the text of **letters/file1** is included. (You can, of course, use the **~p** (tilde p) escape to preview the contents before sending your message.)

Incorporating a Message from Your Mailbox into a Reply

```
$ mailx<CR>
mailx version 2.14  2/9/85  Type ? for help.
"usr/mail/roberts": 2 messages 1 new
>N   1 abc        Tue May 1  08:09  8/155  Meeting Notice
     2 hqtrs      Mon Apr 30 16:57  4/127  Schedule
? m jones<CR>
Subject: Hq Schedule<CR>
Here is a copy of the schedule from headquarters...<CR>
~f 2<CR>
Interpolating: 2
(continue)
As you can see, the boss will be visiting our district <CR>
on the 14th and 15th.<CR>
- Robert
~.
EOT
?
```

There are several important points illustrated in this example:

- The sequence begins in command mode, where you read and respond to your incoming mail. Then you switch into input mode by issuing the command **m jones** (meaning send a message to jones).
- The **~f** escape is used in input mode to call in one of the messages in your mailbox and make it part of the outgoing message. The number **2** after the **~f** means message 2 is to be interpolated (read in).
- **mailx** tells you that message 2 is being interpolated and then tells you to continue.
- When you finish creating and sending the message, you are back in command mode, as shown by the ? prompt. You may now do something else in command mode, or exit

mailx by typing **q**.

An alternate command, the **~m** (tilde m) escape, works the way that **~f** does except the read-in message is indented one tab stop. Both the **~m** and **~f** commands work only if you start out in command mode and then enter a command that puts you into input mode. Other commands that work this way will be covered in the section "How to Manage Incoming Mail."

Changing Parts of the Message Header

The header of a **mailx** message has four components:

- subject
- recipient(s)
- copy-to list
- blind-copy list (a list of intended recipients that is not shown on the copies sent to other recipients)

When you enter the **mailx** command followed by a login or an alias you are put into input mode and prompted for the subject of your message. Once you end the subject line by pressing the RETURN key, **mailx** expects you to type the text of the message. If, at any point in input mode, you want to change or supplement some of the header information, there are four tilde escapes that you can use: **~h**, **~t**, **~c**, and **~b**.

~h displays all the header fields: subject, recipient, copy-to list, and blind copy list, with their current values. You can change a current value, add to it, or, by pressing the RETURN key, accept it.

~t lets you add names to the list of recipients. Names can be either login names or aliases.

~c lets you create or add to a copy-to list for the message. Enter either login names or aliases of those to whom a copy of the message should be sent.

~b lets you create or add to a blind-copy list for the message.

All tilde escapes must be in the first position on a line. For the **~t**, **~c** or **~b**, any additional material on the line is taken to be input for the list in question. Any additional material on a line that begins with a **~h** is ignored.

Adding Your Signature

If you want, you can establish two different signatures with the **sign** and **Sign** environment variables. These can be invoked with the **~a** (tilde a) or **~A** (tilde A) escape, respectively. Assume you have set the value Supreme Commander to be called by the **~A** escape. Here's how it would work:

```
$ mailx -s orders all<CR>
Be ready to move out at 0400 hours.<CR>
~A<CR>
Supreme Commander
~.<CR>
EOT
$
```

Having both escapes (**~a** and **~A**) allows you to set up two forms for your signature. However, because the sender's login automatically appears in the message header when the message is read, no signature is required to identify you.

Keeping a Record of Messages You Send

The **mailx** command offers several ways to keep copies of outgoing messages. Two that you can use without setting any special environment variables are the ~**w** (tilde w) escape and the **-F** option on the command line.

The ~w followed by a file name causes the message to be written to the named file. For example:

```
$ mailx bdr<CR>
Subject: Saving Copies<CR>
When you want to save a copy of<CR>
the text of a message, use the tilde w.<CR>
~w savemail
"savemail" 2/71
~.
EOT
$
```

If you now display the contents of **savemail**, you will see this:

```
$ cat savemail<CR>
When you want to save a copy of
the text of a message, use the tilde w.
$
```

The drawback to this method, as you can see, is that none of the header information is saved.

Using the **-F** option on the command line does preserve the header information. It works as follows:

```
$ mailx -F -s Savings bdr<CR>
This method appends this message to a
file in my current directory named bdr.
~.
EOT
$
```

We can check the results by looking at the file **bdr**.

```
$ cat bdr<CR>
From: kol  Fri May 2  11:14:45  1986
To: bdr
Subject: Savings

This method appends this message to a
file in my current directory named bdr.
$
```

The **-F** option appends the text of the message to a file named after the first recipient. If you have used an alias for the recipient(s) the alias is first converted into the appropriate login(s) and the first login is used as the file name. As noted above, if you have a file by that name in your current directory, the text of the message is appended to it.

Exiting from mailx

When you have finished composing your message, you can leave **mailx** by typing any of the following three commands:

. tilde period (**~.**) is the standard way of leaving input mode. It also sends the message. If you entered input mode from the command mode of **mailx**, you now return to the command mode (as shown by the ? prompt you receive after typing this command). If you started out in input mode, you now return to the shell (as shown by the shell prompt).

~q tilde q (**~q**) simulates an interrupt. It lets you exit the input mode of **mailx**. If you have entered text for a message, it will be saved in a file called **dead.letter** in your home directory.

~x tilde x (**~x**) simulates an interrupt. It lets you exit the input mode of **mailx** without saving anything.

Summary

In the preceding paragraphs we have described and shown examples of some of the tilde escape commands available when sending messages via the **mailx** command. (See the **mailx**(1) manual page in the *User's Reference Manual*.)

How to Manage Incoming Mail

mailx has over fifty commands which help you manage your incoming mail. See the **mailx**(1) manual page in the *User's Reference Manual* for a list of all of them (and their synonyms) in alphabetic order. The most commonly used commands (and arguments) are described in the following subsections:

- the *msglist* argument
- commands for reading and deleting mail
- commands for saving mail
- commands for replying to mail
- commands for getting out of **mailx**

The msglist Argument

Many commands in **mailx** take a form of the *msglist* argument. This argument provides the command with a list of messages on which to operate. If a command expects a *msglist* argument and you do not provide one, the command is performed on the current message. Any of the following formats can be used for a *msglist*:

n	message number *n* the current message
^	the first undeleted message
$	the last message
*	all messages
n-m	an inclusive range of message numbers
user	all messages from *user*
/string	All messages with *string* in the subject line (case is ignored)
:*c*	all messages of type *c* where *c* is:

d - deleted messages
n - new messages
o - old messages

r - read messages
u - unread messages

The context of the command determines whether this type of specification makes sense.

Here are two examples (the ? is the command mode prompt):

```
? d 1-3          [ Delete messages 1, 2 and 3 ]
? s bdr bdr      [ Save all messages from user bdr in a
                     file named bdr. ]

?
```

Additional examples may be found throughout the next three subsections.

Commands for Reading and Deleting Mail

When a message arrives in your mailbox the following notice appears on your screen:

```
you have mail
```

The notice appears when you log in or when you return to the shell from another procedure.

Reading Mail

To read your mail, enter the **mailx** command with or without arguments. Execution of the command places you in the command mode of **mailx**. The next thing that appears on your screen is a display that looks something like this:

```
mailx version 2.14  10/19/86    Type ? for help
"/usr/mail/bdr":   3 messages  3 new
> N 1 rbt          Thur Apr 30 14:20   8/190  Review Session
  N 2 admin        Thur Apr 30 15:56   5/84   New printer
  N 3 daves        Fri  May  1 08:39  64/1574 Reorganization
?
```

The first line identifies the version of **mailx** used on your system, displays the date, and reminds you that help is available by typing a question mark (**?**). The second line shows the path name of the file used as input to the display (the file name is normally the same as your login name) together with a count of the total number of messages and their status. The rest of the display is header information from the incoming messages. The messages are numbered in sequence with the last one received at the bottom of the list. To the left of the numbers there may be a status indicator; N for new, U for unread. A greater than sign (>) points to the current message. Other fields in the header line show the login of the originator of the message, the day, date and time it was delivered, the number of lines and characters in the message, and the message subject. The last field may be blank.

When the header information is displayed on your screen, you can print messages either by pressing the RETURN key or entering a command followed by a *msglist* argument. If you enter a command with no *msglist* argument, the command acts on the message pointed at by the > sign. Pressing the RETURN key is the equivalent of a typing the **p** (for print) command without a *msglist* argument; the message displayed is the one pointed at by the > sign. To read some other message (or several others in succession), enter a **p** (for print) or **t** (for type) followed by the message number(s). Here are some examples:

```
? <CR>          [ Print the current message. ]
? p 2<CR>       [ Print message number 2.    ]
? p daves<CR>   [ Print all messages from user daves. ]
```

The command **t** (for type) is a synonym of **p** (for print).

Scanning Your Mailbox

The **mailx** command lets you look through the messages in your mailbox while you decide which ones need your immediate attention.

When you first enter the **mailx** command mode, the banner tells you how many messages you have and displays the header line for twenty messages. (If you are dialed into the computer system, only the header lines for ten messages are displayed.) If the total number of messages exceeds one screenful, you can display the next screen by entering the **z** command. Typing **z-** causes a previous screen (if there is one) to be displayed. If you want to see the header information for a specific group of messages, enter the **f** (for from) command followed by the *msglist* argument.

Here are examples of those commands:

? **z**	[*Scroll forward one screenful of header lines.*]
? **z-**	[*Scroll backward one screenful.*]
? **f daves**	[*Display headers of all messages from user daves.*]

Switching to Other Mail Files

When you enter **mailx** by issuing the command:

$ **mailx**<CR>

you are looking at the file /**usr**/**mail**/*your_login*.

mailx lets you switch to other mail files and use any of the **mailx** commands on their contents. (You can even switch to a non-mail file, but if you try to use **mailx** commands you are told `No applicable messages`.) The switch to another file is done with the **fi** or **fold** command (they are synonyms) followed by the *filename*. The following special characters work in place of the *filename* argument:

%	the current mailbox
%*login*	the mailbox of the owner of *login* (if you have the required permissions)
#	the previous file
&	the current mbox

Here is an example of how this might look on your screen:

```
$ mailx<CR>

mailx version 2.14 10/19/86  Type ? for help.
"usr/mail/daves":  3 messages 2 new 3 unread
  U 1 jaf          Sat May 9 07:55   7/137   test25
> N 2 todd         Sat May 9 08:59   9/377   UNITS requiremts
  N 3 has          Sat May 9 11:08  29/1214  access to bailey

? fi &       [ Enter this command to transfer to your mbox. ]

Held 3 messages in /usr/mail/daves
"/fs1/daves/mbox":  74 messages 10 unread
  .
  .
  .
? q<CR>
$
```

Deleting Mail

To delete a message, enter a **d** followed by a *msglist* argument. If the *msglist* argument is omitted, the current message is deleted. The messages are not deleted until you leave the mailbox file you are processing. Prior to that, the **u** (for undelete) gives you the opportunity to change your mind. Once you have issued the quit command (**q**) or switched to another file, however, the deleted messages are gone.

mailx permits you to combine the delete and print command and enter a **dp**. This is like saying, "Delete the message I just read and show me the next one." Here are some examples of the delete command:

```
? d *      [ Delete all my messages. ]
? d r      [ Delete all messages that have been read. ]
? dp       [ Delete current message and print next one. ]
? d 2-5    [ Delete messages 2 through 5. ]
```

Commands for Saving Mail

All messages not specifically deleted are saved when you quit **mailx**. Messages that have been read are saved in a file in your home directory called **mbox**. Messages that have not been read are held in your mailbox (/**usr**/**mail**/*your_login*).

The command to save messages comes in two forms: with an upper case or a lower case **s**. The syntax for the upper case version is:

S [*msglist*]

Messages specified by the *msglist* argument are saved in a file in the current directory named for the author of the first message in the list.

The syntax for the lower case version is:

s [*msglist*] [*filename*]

Messages specified by the *msglist* argument are saved in the file named in the *filename* argument. If you omit the *msglist* argument, the current message is saved. If you are using logins for file names, this can lead to some ambiguity. If **mailx** is puzzled, you will get an error message.

Commands for Replying to Mail

The command for replying to mail comes in two forms: with an upper case or a lower case **r**. The principal difference between the two forms is that the upper case form (**R**) causes your response to be sent only to the originator of the message, while the lower case form (**r**) causes your response to be sent not only to the originator but also to all other recipients. (There are other differences between these two forms. For details, see the **mailx**(1) manual page in the *User's Reference Manual*.)

When you reply to a message, the original subject line is picked up and used as the subject of your reply. Here's an example of the way it looks:

```
$ mailx<CR>

mailx version 2.14 10/19/83  Type ? for help.
"usr/mail/daves":  3 messages 2 new 3 unread
  U 1 jaf          Wed May 9 07:55   7/137   test25
> N 2 todd         Wed May 9 08:59   9/377   UNITS requiremts
  N 3 has          Wed May 9 11:08  29/1214  access to bailey

? R 2
To: todd
Subject: Re: UNITS requirements
```

Assuming the message about UNITS requirements had been sent to some additional people, and the lower case **r** had been used, the header might have appeared like this:

```
? r 2
To: todd eg has jcb bdr
Subject: Re: UNITS requirements
```

Commands for Getting Out of mailx

There are two standard ways of leaving **mailx**: with a **q** or with an **x**. If you leave **mailx** with a **q**, you see messages that summarize what you did with your mail. They look like this:

```
? q<CR>
Saved 1 message in /fs1/bdr/mbox
Held 1 message in /usr/mail/bdr
$
```

From the example we can surmise that user **bdr** had at least two messages, read one and either left the other unread or issued a command asking that it be held in **/usr/mail/bdr**. If there were more than two messages, the others were deleted or saved in other files. **mailx** does not issue a message about those.

If you leave **mailx** with an **x**, it is almost as if you had never entered. Mail read and messages deleted are retained in your mailbox. However, if you have saved messages in other files, that action has already taken place and is not undone by the **x**.

mailx Command Summary

In the preceding subsections we have described some of the most frequently used **mailx** commands. (See the **mailx**(1) manual page in the *User's Reference Manual* for a complete list.) If you need help while you are in the command mode of **mailx**, type either a **?** or **help** after the **?** prompt. A list of **mailx** commands and what they do will be displayed on your terminal screen.

The .mailrc File

The **.mailrc** file contains commands to be executed when you invoke **mailx**.

There may be a system-wide start-up file (**/usr/lib/mailx/mailx.rc**) on your system. If it exists it is used by the system administrator to set common variables. Variables set in your **.mailrc** file take precedence over those in **mailx.rc.**

Most **mailx** commands are legal in the **.mailrc** file. However, the following commands are NOT legal entries:

! (or) **shell**	escape to the shell
Copy	save messages in *msglist* in a file whose name is derived from the author
edit	invoke the editor
visual	invoke vi
followup	respond to a message
Followup	respond to a message, sending a copy to *msglist*
mail	switch into input mode
reply	respond to a message
Reply	respond to the author of each message in *msglist*

You can create your own **.mailrc** with any editor, or copy a friend's. Figure 8-5 shows a sample **.mailrc** file.

```
if r
                        cd $HOME/mail
endif
set allnet append asksub askcc autoprint dot
set metoo quiet save showto header hold keep keepsave
set outfolder
set folder='mail'
set record='outbox'
set crt=24
set EDITOR='/bin/ed'
set sign='Roberts'
set Sign='Jackson Roberts, Supervisor'
set toplines=10
alias fred              fjs
alias bob               rcm
alias alice             ap
alias mark              mct
alias donna             dr
alias pat               pat
group robertsgrp        fred bob alice pat mark
group accounts          robertsgrp donna
```

Figure 8-5: Sample **.mailrc** File

The example in Figure 8-5 includes the commands you are most likely to find useful: the **set** command and the **alias** or **group** commands.

The **set** command is used to establish values for environment variables. The command syntax is:

set
set *name*
set *name* = *string*
set *name* = *number*

When you issue the **set** command without any arguments, **set** produces a list of all defined variables and their values. The argument *name* refers to an environmental variable. More than one *name* can be entered after the **set** command. Some variables take a string or numeric value. String values are enclosed in single quotes.

When you put a value in an environment variable by making an assignment such as **HOME**=*my_login*, you are telling the shell how to interpret that variable. However, this type of assigment in the shell does not make the value of the variable accessible to other UNIX system programs that need to reference environment variables. To make it accessible, you must export the variable. If you set the **TERM** variable in your environment in Chapter 6 or Chapter 7, you will remember using the **export** command as shown in the following example:

```
$ TERM=5425
$ export TERM
```

When you export variables from the shell in this way, programs that reference environment variables are said to import them. Some of these variables (such as **EDITOR** and **VISUAL**) are not peculiar to **mailx**, but may be specified as general environment variables and imported from your execution environment. If a value is set in **.mailrc** for an imported variable it overrides the imported value. There is an **unset** command, but it works only against variables set in **.mailrc**; it has no effect on imported variables.

There are forty-one environment variables that can be defined in your **.mailrc**; too many to be fully described in this document. For complete information, consult the **mailx**(1) manual page in the *User's Reference Manual*.

Three variables used in the example in Figure 8-5 deserve special attention because they demonstrate how to organize the filing of messages. These variables are: **folder**, **record**, and **outfolder**. All three are interrelated and control the directories and files in which copies of messages are kept.

To put a value into the **folder** variable, use the following format:

set folder = *directory*

This specifies the directory in which you want to save standard mail files. If the directory name specified does not begin with a / (slash), it is presumed to be relative to **$HOME**. If **folder** is an exported shell variable, you can specify file names (in commands that call for a *filename* argument) with a / before the name; the name will be expanded so that the file is put into the **folder** directory.

To put a value in the **record** variable, use the following format:

set record = *filename*

This directs **mailx** to save a copy of all outgoing messages in the specified file. The header information is saved along with the text of the message. By default, this variable is disabled.

The **outfolder** variable causes the file in which you store copies of outgoing messages (enabled by the variable **record** =) to be located in the **folder** directory. It is established by being named in a **set** command. The default is **nooutfolder**.

The **alias** and **group** commands are synonyms. In Figure 8-5, the **alias** command is used to associate a name with a single login; the **group** command is used to specify multiple names that can be called in with one pseudonym. This is a nice way to distinguish between single and group aliases, but if you want, you can treat the commands as exact equivalents. Notice, too, that aliases can be nested.

In the **.mailrc** file shown in Figure 8-5, the alias **robertsgroup** represents five users; three of them are specified by previously defined aliases and one is specified by a login. The fifth user, **pat**, is specified by both a login and an alias. The next group command in the example, **accounts**, uses the alias **robertsgroup** plus the alias **donna**. It expands to twelve logins.

The **.mailrc** file in Figure 8-5 includes an **if-endif** command. The full syntax of that command is:

```
if s | r mail_commands

  else mail_commands

endif
```

The **s** and **r** stand for send and receive, so you can cause some initializing commands to be executed according to whether **mailx** is entered in input mode (send) or command mode (receive). In the preceding example, the command is issued to change directory to **$HOME/mail** if reading mail. The user in this case had elected to set up a subdirectory for handling incoming mail.

The environment variables shown in this section are those most commonly included in the **.mailrc** file. You can, however, specify any of them for one session only whenever you are in command mode. For a complete list of the environment variables you can set in **mailx** see the **mailx**(1) manual page in the *User's Reference Manual*.

Sending and Receiving Files

This section describes the commands available for transferring files: the **mail** command for small files (a page or less), and the **uucp** and **uuto** commands for long files. The **mail** command can be used for transferring a file either within a local system or to a remote system. The **uucp** and **uuto** commands transfer files from one system to another.

Sending Small Files: the mail Command

To send a file in a **mail** message, you must redirect the input to that file on the command line. Use the < (less than) redirection symbol as follows:

mail *login* < *filename* <**CR**>

(For further information on input redirection, see Chapter 7.) Here *login* is the recipient's login ID and *filename* is the name of the file you want to send. For example, to send a copy of a file called **agenda** to the owner of login **sarah** (on your system) type the following command line:

```
$ mail sarah < agenda <CR>
$
```

The prompt that appears on the second line means the contents of **agenda** have been sent. When **sarah** issues the **mail** command to read her messages, she will receive **agenda**.

To send the same file to more than one user on your system, use the same command line format with one difference; in place of one login ID, type several, separated by spaces. For example:

```
$ mail sarah tommy dingo wombat < agenda <CR>
$
```

Again, the prompt returned by the system in response to your command is a signal that your message has been sent.

The same command line format, with one addition, can also be used to send a file to a user on a remote system that can communicate with yours. In this case, you must specify the name of the remote system before the user's login name. Separate the system name and the login name with an **!** (exclamation point):

mail *system!login* < *filename* <**CR**>

For example:

```
$ mail dumbo!wombat < agenda<CR>
$
```

The system prompt on the second line means that your message (containing the file) has been queued for sending.

If you are using **mailx**, you cannot use the **mail** command line syntax to send a file. Instead, you use the **~r** option as follows:

```
$ mailx phyllis
Subject: Memo
~r memo
$
```

Sending Large Files

The **uucp** and **uuto** commands allow you to transfer files to a remote computer. **uucp** allows you to send files to the directory of your choice on the destination system. If you are transferring a file to a directory that you own, you will have permission to put the file in that directory. (See Chapter 3 for information on directory and file permissions.) However, if you are transferring the file to another user's directory, you must be sure, in advance, that the user has given you permission to write a file to his or her directory.

In addition, because you must specify path names that are often long and accuracy is required, **uucp** command lines may be cumbersome and lead to error.

The **uuto** command is an enhanced version of **uucp**. It automatically sends files to a public directory on the recipient's system called **/usr/spool/uucppublic**. This means you cannot choose a destination file. However, it also means that you can transfer a file at any time without having to request write permission from the owner of the destination directory. Finally, the **uuto** command line is shorter and less complicated than the **uucp** command line. When you type a **uuto** command line, the likelihood of making an error is greatly reduced.

Getting Ready: Do You Have Permission?

Before you actually send a file with the **uucp** or **uuto** command, you need to find out whether or not the file is transferable. To do that, you must check the file's permissions. If they are not correct, you must use the **chmod** command to change them, if you own the files. (Permissions and the **chmod** command are covered in Chapter 3.)

There are two permission criteria that must be met before a file can be transferred using **uucp** or **uuto**.

- The file to be transferred must have read permission (**r**) for others.
- The directory that contains the file must have read (**r**) and execute (**x**) permission for others.

For example, assume that you have a file named **chicken**, under a directory named **soup** (in your home directory). You want to send a copy of the **chicken** file to another user with the **uuto** command. First, check the permissions on **soup**:

```
$ ls -l <CR>
total 4
drwxr-xr-x  2   reader  group1    45    Feb 9   10:43   soup
$
```

The response of the **ls** command shows that **soup** has read (**r**) and execute (**x**) permissions for all three groups; no changes have to be made. Now use the **cd** command to move from your home directory to **soup**, and check the permissions on the file **chicken**:

```
$ ls -l chicken <CR>
total 4
-rw-------  1   reader  group1  3101    Mar 1   18:22   chicken
$
```

The command's output means that you (the user) have permission to read the file **chicken**, but no one else does. To add read permissions for your group (**g**) and others (**o**), use the **chmod** command:

$ **chmod go+r chicken <CR>**

Now check the permissions again with the **ls -l** command:

```
$ ls -l chicken<CR>
total 4
-rw-r--r--  1  reader  group1  3101  Mar01  18:22  chicken
$
```

This confirms that the file is now transferable; you can send it with the **uucp** or **uuto** command. After you send copies of the file, you can reverse the procedure and replace the previous permissions.

The uucp Command

The command **uucp** (short for UNIX-to-UNIX system copy) allows you to copy a file directly to the home directory of a user on another computer, or to any other directory you specify and for which you have write permission.

uucp is not an interactive command. It performs its work silently, invisible to the user. Once you issue this command you may run other processes.

Transferring a file between computers is a multiple-step procedure. First, a work file, containing instructions for the file transfer, must be created. When requested, a data file (a copy of the file being sent) is also made. Then the file is ready to be sent. When you issue the **uucp** command, it performs the preliminary steps described above (creating the necessary files in a dedicated directory called a **spool** directory), and then calls the **uucico** daemon that actually transfers the file. (Daemons are system processes that run in background.) The file is placed in a queue and **uucico** sends it at the first available time.

Thus, the **uucp** command allows you to transfer files to a remote computer without knowing anything except the name of the remote computer and, possibly, the login ID of the remote user(s) to whom the file is being sent.

Command Line Syntax

uucp allows you to send:

- one file to a file or a directory or
- multiple files to a directory

To deliver your file(s), **uucp** must know the full path name of both the *source-file* and the *destination-file*. However, this does not mean you must type out the full path name of both files every time you use the **uucp** command. There are several abbreviations you can use once you become familiar with their formats; **uucp** will expand them to full path names.

To choose the appropriate designations for your *source-file* and *destination-file*, begin by identifying the *source-file*'s location relative to your own current location in the file system. (We'll assume, for the moment, that the *source-file* is in your local system.) If the *source-file* is in your current directory, you can specify it by its name alone (without a path). If the *source-file* is not in your current directory, you must specify its full path name.

How do you specify the *destination-file*? Because it is on a remote system, the *destination-file* must always be specified with a path name that begins with the name of the remote system. After that, however, **uucp** gives you a choice: you can specify the full path or use either of two forms of abbreviation. Your *destination-file* should have one of the following three formats:

- *system_name*!*full_path*
- *system_name*!~*login_name*[/*directory_name*/*filename*]
- *systemname*!~/*login_name*[/directory_name/filename]

The login name, in this case, belongs to the recipient of the file.

Until now we have described what to do when you want to send a file from your local system to a remote system. However, it is also possible to use **uucp** to send a file from a remote system to your local system. In either case, you can use the formats described above to specify either *source-files* or *destination-files*. The important distinction in choosing one of these formats is not whether a file is a *source-file* or a *destination-file*, but where you are currently located in the file system relative to the files you are specifying. Therefore, in the formats shown above, the *login_name* could refer to the login of the owner or the recipient of either a *source-file* or a *destination-file*.

For example, let's say you are login **kol** on a system called **mickey**. Your home directory is **/usr/kol** and you want to send a file called **chap1** (in a directory called **text** in your home directory) to login **wsm** on a system called **minnie**. You are currently working in **/usr/kol/text**, so you can specify the *source-file* with its relative path name, **chap1**. Specify the *destination-file* in any of the ways shown in the following command lines:

- Specify the *destination-file* with its full path name:

  uucp chap1 minnie!/usr/wsm/receive/chap1

- Specify the *destination-file* with *~login_name* (which expands to the name of the recipient's home directory) and a name for the new file.

  uucp chap1 minnie!~wsm/receive/chap1

  (The file will go to **minnie!/usr/wsm/receive/chap1**.)

- Specify the *destination-file* with *~login_name* (which expands to the recipient's home directory) but without a name for the new file; **uucp** will give the new file the same name as the *source-file*.

  uucp chap1 minnie!~wsm/receive

  (The file will go to **minnie!/usr/wsm/receive/chap1**.)

- Specify the *destination-file* with *~/login_name*. This expands to the recipient's subdirectory in the public directory on the remote system.

uucp chap1 minnie!~/wsm

(The file will go to **minnie!/usr/usr/spool/uucppublic/wsm**)

Sample Usage of Options with the uucp Command

Suppose you want to send a file called **minutes** to a remote computer named **eagle**. Enter the command line shown in the following screen:

```
$ uucp -m -s status -j minutes eagle!/usr/gws/minutes <CR>
eagleN3f45
$
```

This sends the file **minutes** (located in your current directory on your local computer) to the remote computer **eagle**, and places it under the path name **/usr/gws** in a file named **minutes**. When the transfer is complete, the user **gws** on the remote computer is notified by mail.

The **-m** option ensures that you (the sender) are also notified by mail as to whether or not the transfer has succeeded. The **-s** option, followed by the name of the file (**status**), asks the program to put a status report of the file transfer in the specified file (**status**).

NOTE: Be sure to include a file name after the **-s** option. If you do not, you will get this message: `uucp failed completely.`

The job ID (`eagleN3f45`) is displayed in response to the **-j** option.

Even if **uucp** does not notify you of a successful transfer soon after you send a file, do not assume that the transfer has failed. Not all systems equipped with networking software have the hardware needed to call other systems. Files being transferred from these so called passive systems must be collected periodically by active systems equipped with the required hardware (see "How the **uucp** Command Works" for details). Therefore, if you are transferring files from a passive system, you may experience some delay. Check with your system administrator to find out whether your system is active or passive.

The previous example uses a full path name to specify the *destination-file*. There are two other ways the *destination-file* can be specified:

- The login directory of **gws** can be specified through use of the (tilde), as shown below:

  eagle!gws/minutes

  is interpreted as:

  eagle!/usr/gws/minutes

- The **uucppublic** area is referenced by a similar use of the tilde prefix to the path name. For example:

  eagle!/gws/minutes

  is interpreted as:

  /usr/spool/uucppublic/gws/minutes

How the uucp Command Works

This section is an overview of what happens when you issue the **uucp** command. An understanding of the processes involved may help you to be aware of the command's limitations and requirements: why it can perform some tasks and not others, why it performs tasks when it does, and why you may or may not be able to use it for tasks that **uucp** performs. For further details see the *System Administrator's Guide* and the *System Administrator's Reference Manual*.

When you enter a **uucp** command, the **uucp** program creates a work file and usually a data file for the requested transfer. (**uucp** does not create a data file when you use the **-c** option.) The work file contains information required for transferring the file(s). The data file is simply a copy of the specified source file. After these files are created in the spool directory, the **uucico** daemon is started.

The **uucico** daemon attempts to establish a connection to the remote computer that is to receive the file(s). It first gathers the information required for establishing a link to the remote computer from the **Systems** file. This is how **uucico** knows what type of device to use in establishing the link. Then **uucico** searches the **Devices** file looking for the devices that match the requirements listed in the **Systems** file. After **uucico** finds an available device, it attempts to establish the link and log in on the remote computer.

When **uucico** logs in on the remote computer, it starts the **uucico** daemon on the remote computer. The two **uucico** daemons then negotiate the line protocol to be used in the file transfer(s). The local **uucico** daemon then transfers the file(s) that you are sending to the remote computer; the remote **uucico** places the file in the specified path name(s) on the remote computer. After your local computer completes the transfer(s), the remote computer may send files that are queued for your local computer. The remote computer can be denied permission to transfer these files with an entry in the **Permissions** file. If this is done, the remote computer must establish a link to your local computer to perform the transfers.

If the remote computer or the device selected to make the connection to the remote computer is unavailable, the request remains queued in the spool directory. Each hour (default), **uudemon.hour** is started by **cron** which in turn starts the **uusched** daemon. When the **uusched** daemon starts, it searches the spool directory for the remaining work files, generates the random order in which these requests are to be processed, and then starts the transfer process (**uucico**) described in the previous paragraphs.

The transfer process described generally applies to an active computer. An active computer (one with calling hardware and networking software) can be set up to poll a passive computer. Because it has networking software, a passive computer can queue file transfers. However, it cannot call the remote computer because it does not have the required hardware. The **Poll** file (**/usr/lib/uucp/Poll**) contains a list of computers that are to be polled in this manner.

Figure 8-6 summarizes the syntax and capabilities of the **uucp** command.

Command Recap		
uucp - copies a file from one computer to another		
command	*options*	*arguments*
uucp	**-j1**, **-m**, **-s** and others*	*source-file*
Description:	**uucp** performs preliminary tasks required to copy a file from one computer to another, and calls **uucico**, the daemon (background process) that transfers the file. The user need only issue the **uucp** command for a file to be copied.	
Remarks:	By default, the only directory to which you can write files is **/usr/spool/uucppublic**. To write to directories belonging to another user, you must receive write permission from that user. Although there are several ways of representing path names as arguments, it is recommended that you type full path names to avoid confusion.	

Figure 8-6: Summary of the **uucp** Command

* See the **uucp**(1) manual page in the *User's Reference Manual* for all available options and an explanation of their capabilities.

The uuto Command

The **uuto** command allows you to transfer files to the public directory of another system. The basic format for the **uuto** command is:

uuto *filename system!login* <**CR**>

where *filename* is the name of the file to be sent, *system* is the recipient's system, and *login* is the recipient's login name.

If you send a file to someone on your local system, you may omit the system name and use the following format:

uuto *filename login* <**CR**>

Sending a File: the -m Option and uustat Command

Now that you know how to determine if a file is transferable, let's take an example and see how the whole thing works.

The process of sending a file by **uuto** is referred to as a job. When you issue a **uuto** command, your job is not sent immediately. First, the file is stored in a queue (a waiting line of jobs) and assigned a job number. When the job's number comes up, the file is transmitted to the remote system and placed in a public directory there. The recipient is notified by a **mail** message and must use the **uupick** command (discussed later in the chapter) to retrieve the file.

For the following discussions, assume this information:

wombat	your login name
sys1	your system name
marie	recipient's login name
sys2	recipient's system name
money	file to be sent

Also assume that the two systems can communicate with each other.

To send the file **money** to login **marie** on system **sys2**, enter the following:

```
$ uuto money sys2!marie <CR>
$
```

The prompt on the second line is a signal that the file has been sent to a job queue. The job is now out of your hands; all you can do is wait for confirmation that the job reached its destination.

How do you know when the job has been sent? The easiest method is to alter the **uuto** command line by adding a **-m** option, as follows:

```
$ uuto -m money sys2!marie<CR>
$
```

This option sends a **mail** message back to you when the job has reached the recipient's system. The message may look something like this:

```
$ mail<CR>
From uucp Thur Apr3 09:45 EST 1986
file /sys1/wombat/money, system sys1
copy succeeded
?
```

If you would like to check if the job has left your system, you can use the **uustat** command. This command keeps track of all the **uucp** and **uuto** jobs you submit and reports the status of each on demand. For example:

```
$ uustat<CR>
1145 wombat sys2 10/05-09:31 10/05-09:33 JOB IS QUEUED
$
```

The elements of this sample status message are as follows:

- 1145 is the job number assigned to the job of sending the file **money** to **marie** on **sys2**.
- **wombat** is the login name of the person requesting the job.
- sys2 is the recipient's system.
- 10/05-09:31 is the date and time the job was queued.
- 10/05-09:33 is the date and time this **uustat** message was sent.
- The final part is a status report on the job. Here the report shows that the job has been queued, but has not yet been sent.

To receive a status report on only one **uuto** job, use the **-j** option and specify the job number on the command line:

uustat -j*jobnumber* **<CR>**

For example, to get a report on the job described in the previous example, specify 1145 (the job number) after the **-j** option:

```
$ uustat -j1145<CR>
1145 wombat sys2 10/05-09:31 10/05-09:37 COPY FINISHED,
JOB DELETED
$
```

This status report shows that the job was sent and deleted from the job queue; it is now in the public directory of the recipient's system. Other status messages and options for the **uustat** command are described in the *User's Reference Manual*.

That is all there is to sending files. To practice, try sending a file to yourself.

Figures 8-7 and 8-8 summarize the syntax and capabilities of the **uuto** and **uustat** commands, respectively.

Command Recap **uuto** - sends files to another login		
command	*options*	*arguments*
uuto	**–m** and others*	*file system*!**login**
Description:	**uuto** sends a specified file to the public directory of a specified system, and notifies the intended recipient (by mail addressed to his or her login) that the file has arrived there.	
Remarks:	Files to be sent must have read permission for others; the file's parent directory must have read and execute permissions for others. The **-m** option notifies the sender by mail when the file has arrived at its destination.	

Figure 8-7: Summary of the **uuto** Command

* See the **uuto**(1) manual page in the *User's Reference Manual* for all available options and an explanation of their capabilities.

Command Recap **uustat** - checks job status of a **uucp** or **uuto** job		
command	*options*	*arguments*
uustat	**-j** and others*	none
Description:	**uustat** reports the status of all **uucp** and **uuto** jobs you have requested.	
Remarks:	The **-j** option, followed by a job number, allows you to request a status report on only the specified job.	

Figure 8-8: Summary of the **uustat** Command

* See the **uustat**(1) manual page in the *User's Reference Manual* for all available options and an explanation of their capabilities.

Receiving Files Sent with uuto: the uupick Command

When a file sent by **uuto** reaches the public directory on your UNIX system, you receive a **mail** message. To continue the previous example, the owner of login **marie** receives the following mail message when the file **money** has arrived in her system's public directory:

```
$ mail
From uucp Wed May 14 09:22 EST 1986
/usr/spool/uucppublic/receive/marie/sys1//money from
sys1!wombat arrived
$
```

The message contains the following pieces of information:

- The first line tells you when the file arrived at its destination.
- The second line, up to the two slashes (//), gives the path name to the part of the public directory where the file has been stored.
- The rest of the line (after the two slashes) gives the name of the file and the sender.

Once you have disposed of the **mail** message, you can use the **uupick** command to store the file where you want it. Type the following command after the system prompt:

uupick <CR>

The command searches the public directory for any files sent to you. If it finds any, it reports the filename(s). It then prints a ? prompt as a request for further instructions from you.

For example, say the owner of login **marie** issues the **uupick** command to retrieve the **money** file. The command will respond as follows:

```
$ uupick <CR>
from system sys1: file money
?
```

There are several available responses; we will look at the most common responses and what they do.

The first thing you should do is move the file from the public directory and place it in your login directory. To do so, type an **m** after the question mark:

```
?
m<CR>
$
```

This response moves the file into your current directory. If you want to put it in some other directory instead, follow the **m** response with the directory name:

```
?
m other_directory<CR>
```

If there are other files waiting to be moved, the next one is displayed, followed by the question mark. If not, **uucpick** returns a prompt.

If you do not want to do anything to that file now, press the RETURN key after the question mark:

```
?
<CR>
```

The current file remains in the public directory until the next time you use the **uupick** command. If there are no more messages, the system returns a prompt.

If you already know that you do not want to save the file, you can delete it by typing **d** after the question mark:

```
?
d<CR>
```

This response deletes the current file from the public directory and displays the next message (if there is one). If there are no additional messages about waiting files, the system returns a prompt.

Finally, to stop the **uupick** command, type a **q** after the question mark:

```
?
q<CR>
```

Any unmoved or undeleted files will wait in the public directory until the next time you use the **uupick** command.

Other available responses are listed in the *User's Reference Manual*.

Figure 8-9 summarizes the syntax and capabilities of the **uupick** command.

Command Recap **uupick** - searches for files sent by **uuto** or **uucp**		
command	*options*	*arguments*
uupick	-s	system name
Description:	**uupick** searches the public directory of your system for files sent by **uuto** or **uucp**. If any are found, the command displays information about the file and prompts you for a response.	
Remarks:	The question mark (?) at the end of the message shows that a response is expected. A complete list of responses is given in the *User's Reference Manual*.	

Figure 8-9: Summary of the **uupick** Command

Networking

Networking is the process of linking computers and terminals so that users may be able to:

- log in on a remote computer as well as a local one
- log in and work on two computers in one work session (without alternately logging off one and logging in on the other)
- exchange data between computers

The commands presented in this section make it possible for you to perform these tasks. The **ct** command allows you to connect your computer to a remote terminal that is equipped with a modem. The **cu** command enables you to connect your computer to a remote computer, and the **uux** command lets you run commands on a remote system, without being logged in on it.

NOTE: On some small computers, the presence of these commands may depend on whether or not networking software is installed. If it is not installed on your system, you will receive a message such as the following when you type a networking command:

```
cu: not found
```

Check with your system administrator to verify the availability of networking commands on your UNIX system.

Connecting a Remote Terminal: the ct Command

The **ct** command connects your computer to a remote terminal equipped with a modem, and allows a user on that terminal to log in. To do this, the command dials the phone number of the modem. The modem must be able to answer the call automatically. When **ct** detects that the call has been answered, it issues a

getty (login) process for the remote terminal and allows a user on it to log in on the computer.

This command can be useful when issued from the opposite end, that is, from the remote terminal itself. If you are using a remote terminal that is far from your computer and want to avoid long distance charges, you can use **ct** to have the computer place a call to your terminal. Simply call the computer, log in, and issue the **ct** command. The computer will hang up the current line and call your (remote) terminal back.

If **ct** cannot find an available dialer, it tells you that all dialers are busy and asks if it should wait until one becomes available. If you answer yes, it asks how long (in minutes) it should wait for one.

Command Line Format

To execute the **ct** command, follow this format:

ct [*options*] *telno* **<CR>**

The argument *telno* is the telephone number of the remote terminal.

Sample Command Usage

Suppose you are logged in on a computer through a local terminal and you want to connect a remote terminal to your computer. The phone number of the modem on the remote terminal is 932-3497. Enter this command line:

ct -h -w5 -s1200 9=9323497<CR>

<u>NOTE:</u> The equal sign (=) represents a secondary dial tone, and dashes (-) following the phone number represent delays (the dashes are useful following a long distance number).

ct will call the modem, using a dialer operating at a speed of 1200 baud. If a dialer is not available, the **-w5** option will cause **ct** to wait for a dialer for five minutes before quitting. The **-h** option tells

ct not to disconnect the local terminal (the terminal on which the command was issued) from the computer.

Now imagine that you want to log in on the computer from home. To avoid long distance charges, use **ct** to have the computer call your terminal:

ct -s1200 9=9323497<CR>

Because you did not specify the **-w** option, if no device is available, **ct** sends you the following message:

```
1 busy dialer at 1200 baud Wait for dialer?
```

If you type **n** (no), the **ct** command exits. If you type **y** (yes), **ct** prompts you to specify how long **ct** should wait:

```
Time, in minutes?
```

If a dialer is available, **ct** responds with:

```
Allocated dialer at 1200 baud
```

This means that a dialer has been found. In any case, **ct** asks if you want the line connecting your remote terminal to the computer to be dropped:

```
Confirm hangup?
```

If you type **y** (yes), you are logged off and **ct** calls your remote terminal back when a dialer is available. If you type **n** (no), the **ct** command exits, leaving you logged in on the computer.

Figure 8-10 summarizes the syntax and capabilities of the **ct** command.

Command Recap **ct** - connect computer to remote terminal		
command	*options*	*arguments*
ct	**-h, -w, -s** and others*	*telno*
Description:	**ct** connects the computer to a remote terminal and allows a user to log in from that terminal.	
Remarks:	The remote terminal must have a modem capable of answering phone calls automatically.	

Figure 8-10: Summary of the **ct** Command

* See the **ct**(1) manual page in the *User's Reference Manual* for all available options and an explanation of their capabilities.

Calling Another UNIX System: the cu Command

The **cu** command connects a remote computer to your computer and allows you to be logged in on both computers simultaneously. This means that you can move back and forth between the two computers, transferring files and executing commands on both, without dropping the connection.

The method used by the **cu** command depends on the information you specify on the command line. You must specify the telephone number or system name of the remote computer. If you specify a phone number, it is passed on to the automatic dial modem. If you specify a system name, **cu** obtains the phone number from the **Systems** file. If an automatic dial modem is not used to establish the connection, the line (port) associated with the direct link to the remote computer can be specified on the

command line.

Once the connection is made, the remote computer prompts you to log in on it. When you have finished working on the remote terminal, log off it and terminate the connection by typing <~.>. You will still be logged in on the local computer.

NOTE: The **cu** command is not capable of detecting or correcting errors; data may be lost or corrupted during file transfers. After a transfer, you can check for loss of data by running the **sum** command or the **ls -l** command on the file that was sent and the file that was received. Both of these commands will report the total number of bytes in each file; if the totals match, your transfer was successful. The **sum** command checks more quickly and gives output that is easier to interpret. (See the **sum**(1) and the **ls**(1) manual pages in the *User's Reference Manual* for details.)

Command Line Format

To execute the **cu** command, follow this format:

cu [*options*] *telno* ¦ *systemname* <**CR**>

The components of the command line are:

telno the telephone number of a remote computer

Equal signs (=) represent secondary dial tones and dashes (-) repreent four-second delays.

systemname a system name that is listed in the **Systems** file.

The **cu** command obtains the telephone number and baud rate from the **Systems** file and searches for a dialer. The **-s**, **-n**, and **-l** options should not be used together with *systemname*. (To see the list of computers in the **Systems** file, run the **uuname** command.)

Once your terminal is connected and you are logged in on the remote computer, all standard input (input from the keyboard) is sent to the remote computer. Figures 8-11 and 8-12 show the commands you can execute while connected to a remote computer through **cu**.

String	Interpretation
.	Terminate the link.
!	Escape to the local computer without dropping the link. To return to the remote computer, type <^**d**> (control-d).
!*command*	Execute *command* on the local computer.
$*command*	Run *command* locally and send its output to the remote system.
%cd *path*	Change the directory on the local computer where *path* is the path name or directory name.
%take *from* [*to*]	Copy a file named *from* (on the remote computer) to a file named *to* (on the local computer). If *to* is omitted, the *from* argument is used in both places.
%put *from* [*to*]	Copy a file named *from* (on the local computer) to a file named *to* (on the remote computer). If *to* is omitted, the *from* argument is used in both places.
...	Send a line beginning with (**...**) to the remote computer.
%break	Transmit a BREAK to the remote computer (can also be specified as **%b**).

Figure 8-11: Command Strings for Use with **cu** (Sheet 1 of 2)

String	Interpretation
%nostop	Turn off the handshaking protocol for the remainder of the session. This is useful when the remote computer does not respond properly to the protocol characters.
%debug	Turn the **-d** debugging option on or off (can also be specified as **%d**).
t	Display the values of the terminal I/O (input/output) structure variables for your terminal (useful for debugging).
l	Display the values of the termio structure variables for the remote communication line (useful for debugging).

Figure 8-12: Command Strings for Use with **cu** (Sheet 2 of 2)

NOTE: The use of **%put** requires **stty** and **cat** on the remote computer. It also requires that the current erase and kill characters on the remote computer be identical to the current ones on the local computer.

The use of **%take** requires the existence of the **echo** and **cat** commands on the remote computer. Also, **stty tabs** mode should be set on the remote computer if tabs are to be copied without expansion.

Sample Command Usage

Suppose you want to connect your computer to a remote computer called **eagle**. The phone number for eagle is 847-7867. Enter the following command line:

cu -s1200 9=8477867<CR>

The **-s1200** option causes **cu** to use a 1200 baud dialer to call **eagle**. If the **-s** option is not specified, **cu** uses a dialer at the default speed, 300 baud.

When **eagle** answers the call, **cu** notifies you that the connection has been made, and prompts you for a login ID:

```
connected
login:
```

Enter your login ID and password.

The **take** command allows you to copy files from the remote computer to the local computer. Suppose you want to make a copy of a file named **proposal** for your local computer. The following command copies **proposal** from your current directory on the remote computer and places it in your current directory on the local computer. If you do not specify a file name for the new file, it will also be called **proposal**.

%take proposal<CR>

The **put** command allows you to do the opposite: copy files from the local computer to the remote computer. Say you want to copy a file named **minutes** from your current directory on the local computer to the remote computer. Type:

%put minutes minutes.9-18<CR>

In this case, you specified a different name for the new file (**minutes.9-18**). Therefore the copy of the **minutes** file that is made on the remote computer will be called **minutes.9-18**.

Figure 8-13 summarizes the syntax and capabilities of the **cu** command.

Command Recap **cu** - connects computer to remote computer		
command	*options*	*arguments*
cu	**-s** and others*	*telno* (or) *systemname*
Description:	**cu** connects your computer to a remote computer and allows you to be logged in on both simultaneously. Once you are logged in, you can move between computers to execute commands and transfer files on each without dropping the link.	

Figure 8-13: Summary of the **cu** Command

* See the **cu**(1) manual page in the *User's Reference Manual* for all available options and an explanation of their capabilities.

Executing Commands on a Remote System: the uux Command

The command **uux** (short for UNIX-to-UNIX system command execution) allows you to execute UNIX system commands on remote computers. It can gather files from various computers, execute a command on a specified computer, and send the standard output to a file on a specified computer. The execution of certain commands may be restricted on the remote machine. The command notifies you by mail if the command you have requested is not allowed to execute.

Command Line Format

To execute the **uux** command, follow this format:

uux [*options*] *command-string* **<CR>**

The *command-string* is made up of one or more arguments. All special shell characters (such as "< > ¦ ^") must be quoted either by quoting the entire *command-string* or quoting the character as a separate argument. Within the *command-string* the command and file names may contain a *system name!* prefix. All arguments that do not contain a *systemname* are interpreted as command arguments. A file name may be either a full path name or the name of a file under the current directory (on the local computer).

Sample Command Usage

If your computer is hard-wired to a larger host computer you can use **uux** to get printouts of files that reside on your computer by entering:

pr minutes ¦ uux -p host!lp <CR>

This command line queues the file **minutes** to be printed on the area printer of the computer **host**.

Figure 8-14 summarizes the syntax and capabilities of the **uux** command.

Command Recap **uux** - executes commands on a remote computer		
command	*options*	*arguments*
uux	**-1**, **-p**, and others*	*command-string*
Description:	**uux** allows you to run UNIX system commands on remote computers. It can gather files from various computers, run a command on a specified computer, and send the standard output to a file on a specified computer.	
Remarks:	By default, users of the **uux** command have permission to run only the **mail** and **mailx** commands. Check with your system administrator to find out if users on your system have been granted permission to run other commands.	

Figure 8-14: Summary of the **uux** Command

* See the **uux**(1) manual page in the *User's Reference Manual* for all available options and an explanation of their capabilities.

APPENDICES, GLOSSARY, INDEX

APPENDICES

The UNIX System Files

This appendix summarizes the description of the file system given in Chapter 1 and reviews the major system directories in the **root** directory.

File System Structure

The UNIX System files are organized in a hierarchy; their structure is often described as an inverted tree. At the top of this tree is the root directory, the source of the entire file system. It is designated by a / (slash). All other directories and files descend and branch out from **root**, as shown in Figure A-1.

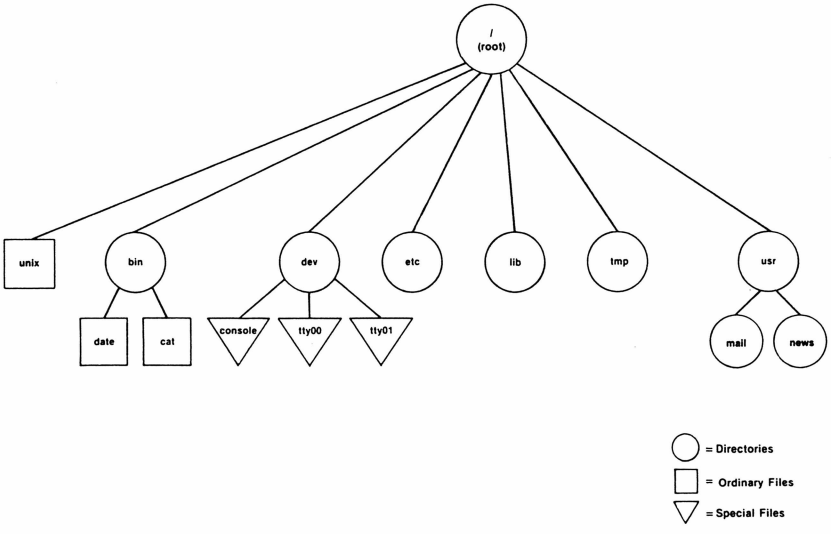

Figure A-1: Directory Tree from **root**

One path from root leads to your home directory. You can organize and store information in your own hierarchy of directories and files under your home directory.

Other paths lead from **root** to system directories that are available to all users. The system directories described in this book are common to all UNIX system installations and are provided and maintained by the operating system.

In addition to this standard set of directories, your UNIX system may have other system directories. To obtain a listing of the directories and files in the **root** directory on your UNIX system, type the following command line:

ls -l / <CR>

To move around in the file structure, you can use path names. For example, you can move to the directory **/bin** (which contains UNIX system executable files) by typing the following command line:

cd /bin <CR>

To list the contents of a directory, issue one of the following command lines:

ls <CR>	for a list of file and directory names
ls -l <CR>	for a detailed list of file and directory names

To list the contents of a directory in which you are not located, issue the **ls** command as shown in the following examples:

ls /bin <CR>	for a short listing
ls -l /bin <CR>	for a detailed listing

The following section provides brief descriptions of the **root** directory and the system directories under it, as shown in Figure A-1.

UNIX System Directories

/	The source of the file system (called **root** directory)
/bin	Contains many executable programs and utilities, such as the following: **cat** **date** **login** **grep** **mkdir** **who**
/lib	Contains available program libraries and language libraries, such as **libc.a** system calls, standard I/O **libm.a** math routines and support for languages such as C, FORTRAN, and BASIC.
/dev	Contains special files that represent peripheral devices, such as: **console** console **lp** line printer **tty***n* user terminal(s) **dsk/*** disks
/etc	Contains programs and data files for system administration
/tmp	Contains temporary files, such as the buffers created for editing a file
/usr	Contains the following subdirectories which, in turn, contain the data listed below: **news** important news items **mail** electronic mail **spool** files waiting to be printed on the line printer

Basic UNIX System Commands

at

Request that a command be run in background mode at a time you specify on the command line. If you do not specify a time, **at**(1) displays the job numbers of all jobs you have running in **at**(1), **batch**(1), or background mode.

A sample format is:

at 8:45am Jun 09 < CR >
command1 **< CR >**
command2 **< CR >**
< ^d >

If you use the **at** command
without the date,
the command executes within twenty-four hours at the time specified.

banner

Display a message (in words up to ten characters long) in large letters on the standard output.

batch

Submit command(s) to be processed when the system load is at an acceptable level. A sample format of this command is:

batch < CR >
command1 **< CR >**
command2 **< CR >**
< ^d >

You can use a shell script for a command in **batch**(1). This may be useful and timesaving if you have a set of commands you frequently submit using this command.

cat Display the contents of a specified file at your terminal. To halt the output on an ASCII terminal temporarily, use <^**s**>; type <^**q**> to restart the output. To interrupt the output and return to the shell on an ASCII terminal, press the BREAK or DELETE key.

cd Change directory from the current one to your home directory. If you include a directory name, changes from the current directory to the directory specified. By using a path name in place of the directory name, you can jump several levels with one command.

cp Copy a specified file into a new file, leaving the original file intact.

cut Cut out specified fields from each line of a file. This command can be used to cut columns from a table, for example.

date Display the current date and time.

diff Compare two files. The **diff**(1) command reports which lines are different and what changes should be made to the second file to make it the same as the first file.

echo Display input on the standard output (the terminal), including the carriage return, and returns a prompt.

ed Edit a specified file using the line editor. If there is no file by the name specified, the **ed**(1) command creates one. See Chapter 5 for detailed instructions on using the **ed**(1) editor.

grep Search a specified file(s) for a specified pattern and prints those lines that contain the pattern. If you name more than one file, **grep**(1) prints the file that contains the pattern.

kill Terminate a background process specified by its process identification number (PID). You can obtain a PID by running the **ps**(1) command.

lex Generate programs to be used in simple lexical analysis of text, perhaps as a first step in creating a compiler. See the *User's Reference Manual* for details.

lp Print the contents of a specified file on a line printer, giving you a paper copy of the file.

lpstat Display the status of any requests made to the line printer. Options are available for requesting more detailed information.

ls List the names of all files and directories except those whose names begin with a dot (**.**). Options are available for listing more detailed information about the files in the directory. (See the **ls**(1) entry in the *User's Reference Manual* for details.)

mail Display any electronic mail you may have received at your terminal, one message at a time. Each message ends with **?** prompt; **mail**(1) waits for you to request an option such as saving, forwarding, or deleting a message. To obtain a list of the available options, type **?**.

When followed by a login name, **mail**(1) sends a message to the owner of that name. You can type as many lines of text as you want. Then type **<^d>** to end the message and send it to the recipient. Press the **BREAK** key to interrupt the mail session.

mailx **mailx**(1) is a more sophisticated, expanded version of electronic mail.

make Maintain and support large programs or documents on the basis of smaller ones. See the **make**(1) page in the *User's Reference Manual* for details.

mkdir Make a new directory. The new directory becomes a subdirectory of the directory in which you issue the **mkdir** command. To create subdirectories or files in the new directory, you must first move into

the new directory with the **cd** command.

mv Move a file to a new location in the file system. You can move a file to a new file name in the same directory or to a different directory. If you move a file to a different directory, you can use the same file name or choose a new one.

nohup Place execution of a command in the background, so it will continue executing after you log off of the system. Error messages are placed in a file called **nohup.out**.

pg Display the contents of a specified file on your terminal, a page at a time. After each page, the system pauses and waits for your instructions before proceeding.

pr Display a partially formatted version of a specified file at your terminal. The **pr**(1) command shows page breaks, but does not implement any macros supplied for text formatter packages.

ps Display the status and number of every process currently running. The **ps**(1) command does not show the status of jobs in the **at**(1) or **batch**(1) queues, but it includes these jobs when they are executing.

pwd Display the full path name of the current working directory.

rm Remove a file from the file system. You can use metacharacters with the **rm**(1) command but should use them with caution; a removed file cannot be recovered easily.

rmdir Remove a directory. You cannot be in the directory you want to delete. Also, the command will not delete a directory unless it is empty. Therefore, you must remove any subdirectories and files that remain in a directory before running this command on it. (See **rm -r** in the *User's Reference Manual* for the ability to remove directories that are not empty.)

sort Sort a file in ASCII order and displays the results on your terminal. ASCII order is as follows:

1. numbers before letters
2. upper case before lower case
3. alphabetical order

There are other options for sorting a file. For a complete list of **sort**(1) options, see the **sort**(1) page in the *User's Reference Manual.*

spell Collect words from a specified file and check them against a spelling list. Words not on the list or not related to words on the list (with suffixes, prefixes, and so on) are displayed.

stty Report the settings of certain input/output options for your terminal. When issued with the appropriate options and arguments, **stty**(1) also sets these input/output option. (See the **stty**(1) entry in the *User's Reference Manual.*)

uname Display the name of the UNIX system on which you are currently working.

uucp Send a specified file to another UNIX system. (See the **uucp**(1) page in the *User's Reference Manual* for details.)

uuname List the names of remote UNIX systems that can communicate with your UNIX system.

uupick Search the public directory for files sent to you by the **uuto**(1) command. If a file is found, **uupick**(1) displays its name and the system it came from, and prompts you (with a **?**) to take action.

uustat Report the status of the **uuto**(1) command you issued to send files to another user.

uuto Send a specified file to another user. Specify the destination in the format *system*!*login*. The *system* must be on the list of systems generated by the **uuname**(1) command.

vi Edit a specified file using the **vi**(1) screen editor. If there is no file by the name you specify, **vi**(1) creates one. (See Chapter 6 for detailed information on using the **vi**(1) editor.)

wc Count the number of lines, words, and characters in a specified file and display the results on your terminal.

who Display the login names of the users currently logged in on your UNIX system. List the terminal address for each login and the time each user logged in.

yacc Impose a structure on the input of a program. See the *User's Reference Manual* for details.

The ed Commands

The general format for **ed** commands is:

[address1,address2]command[parameter]... **<CR>**

where *address1* and *address2* denote line addresses and the *parameters* show the data on which the command operates. The commands appear on your terminal as you type them. You can find complete information on using **ed** commands in Chapter 5, "Line Editor Tutorial."

The following is a glossary of **ed** commands. The commands are grouped according to function.

Commands for Getting Started

ed *filename* Accesses the **ed** line editor to edit a specified file.

a	Appends text after the current line.
.	Ends the text input mode and returns to the command mode.
p	Displays the current line.
d	Deletes the current line.
<CR>	Moves down one line in the buffer.
–	Moves up one line in the buffer.
w	Writes the buffer contents to the file currently associated with the buffer.
q	Ends an editing session. If changes to the buffer were not written to a file, a warning (**?**) is issued. Typing **q** a second time ends the session without writing to a file.

Line Addressing Commands

1, 2, 3...	Denotes line addresses in the buffer.
.	Address of the current line in the buffer.
.=	Displays the current line address.
$	Denotes the last line in the buffer.
,	Addresses the first through the last line.
;	Addresses the current line through the last line.
x	Relative address, determined by adding *x* to the current line number.
−*x*	Relative address, determined by subtracting *x* from the current line number.
/*abc*	Searches forward in the buffer and addresses the first line after the current line that contains the pattern *abc*.
?*abc*	Searches backward in the buffer and addresses the first line before the current line that contains the pattern *abc*.
g/*abc*	Addresses all lines in the buffer that contain the pattern *abc*.
v/*abc*	Addresses all lines in the buffer that do not contain the pattern *abc*.

Display Commands

p	Displays the specified lines in the buffer.
n	Displays the specified lines preceded by their line addresses and a tab space.

Text Input

a Enters text after the specified line in the buffer.

i Enters text before the specified line in the buffer.

c Replaces text in the specified lines with new text.

. When typed on a line by itself, ends the text input mode and returns to the command mode.

Deleting Text

d Deletes one or more lines of text (command mode).

u Undoes the last command given (command mode).

@ Deletes the current line (in text input mode) or a command line (in command mode).

or **BACKSPACE**
Deletes the last character entered as text (in input mode).

Substituting Text

*address1,address2***s***/old_text/new_text/command*
Substitutes *new_text* for *old_text* within the range of lines denoted by *address1,address2* (which may be numbers, symbols, or text). The *command* may be **g**, **l**, **n**, **p**, or **gp**.

Special Characters

. Matches any single character in search or substitution patterns.

***** Matches zero or more occurrences of the preceding character in search or substitution patterns.

[...] Matches the first occurrence of a pattern in the brackets.

[^...]	Matches the first occurrence of a character that is not in the brackets.
.*	Matches zero or more occurrences of any characters following the period in search or substitution patterns.
^	The circumflex (^) matches the beginning of the line in search or substitution patterns.
$	Matches the end of the line in search or substitution patterns.
****	Takes away the special meaning of the special character that follows in search and substitution patterns.
&	Repeats the last pattern to be substituted.
%	Repeats the last replacement pattern.

Text Movement Commands

m	Moves the specified lines of text after a destination line; deletes the lines at the old location.
t	Copies the specified lines of text and places the copied lines after a destination line.
j	Joins the current line with the next contiguous line.
w	Copies (writes) the buffer contents into a file.
r	Reads in text from another file and appends it to the buffer.

Other Useful Commands and Information

h	Displays a short explanation for the preceding diagnostic response (**?**).
H	Turns on the help mode, which automatically displays an explanation for each diagnostic response (**?**) during the editing session.

l	Displays nonprinting characters in the text.
f	Displays the current file name.
f *newfile*	Changes the current file name associated with the buffer to *newfile*.
!*command*	Allows you to escape, temporarily, to the shell to execute a shell command.
ed.hup	If the terminal is hung up before a write command, the editing buffer is saved in the file *ed.hup*.

vi Quick Reference

This appendix is a glossary of commands for the screen editor **vi**. The commands are grouped according to function.

The general format of a **vi** command is:

[*x*][*command*]*text-object*

where *x* denotes a number and *text-object* shows the portion of text on which the command operates. The commands appear on your screen as you type them. For an introduction to the use of **vi** commands, see Chapter 6, "Screen Editor Tutorial."

Commands for Getting Started

Shell Commands

TERM = *code*	Puts a code name for your terminal into the variable **TERM**.
export TERM	Conveys the value of **TERM** (the terminal code) to any UNIX system program that is terminal dependent.
tput init	Initializes the terminal so that it will function properly with various UNIX system programs.

NOTE: Before you can use **vi**, you must complete the first three steps represented by the above three lines: setting the **TERM** variable, exporting the value of **TERM**, and running the **tput init** command.

vi *filename*	Accesses the **vi** screen editor so that you can edit a specified file.

Basic vi Commands

<a>	Enters text input mode and appends text after the cursor.
<ESC>	Escape; leaves text input mode and returns to command mode.
<h>	Moves the cursor to the left one character.
<j>	Moves the cursor down one line in the same column.
<k>	Moves the cursor up one line in the same column.
<l>	Moves the cursor to the right one character.
<x>	Deletes the current character.
<CR>	Carriage return; moves the cursor down to the beginning of the next line.
<ZZ>	Writes changes made to the buffer to the file and quits **vi**.
:w	Writes changes made to the buffer to the file.
:q	Quits **vi** if changes made to the buffer have been written to a file.

Commands for Positioning in the Window

Positioning by Character

<h>	Moves the cursor one character to the left.
<BACKSPACE>	Backspace; moves the cursor one character to the left.

<l>	Moves the cursor one character to the right.
<space bar>	Moves the cursor one character to the right.
<f*x***>**	Moves the cursor right to the specified character *x*.
<F*x***>**	Moves the cursor left to the specified character *x*.
<t*x***>**	Moves the cursor right to the character just before the specified character *x*.
<T*x***>**	Moves the cursor left to the character just after the specified character *x*.
<;>	Continues the search for the character specified by the <**f**>, <**F**>, <**t**>, or <**T**> commands. The **;** remembers the character specified and searches for the next occurrence of it on the current line.
<,>	Continues the search for the character specified by the <**f**>, <**F**>, <**t**>, or <**T**> commands. The **,** remembers the character specified and searches for the previous occurrence of it on the current line.

Positioning by Line

<j>	Moves the cursor down one line from its present position, in the same column.
<k>	Moves the cursor up one line from its present position, in the same column.
< + >	Moves the cursor down to the beginning of the next line.
<CR>	Carriage return; moves the cursor down to the beginning of the next line.

< – >	Moves the cursor up to the beginning of the next line.

Positioning by Word

<w>	Moves the cursor to the right, to the first character in the next word.
<b>	Moves the cursor back to the first character of the previous word.
<e>	Moves the cursor to the end of the current word.

Positioning by Sentence

<(>	Moves the cursor to the beginning of the sentence.
<) >	Moves the cursor to the beginning of the next sentence.

Positioning by Paragraph

<{>	Moves the cursor to the beginning of the paragraph.
<}>	Moves the cursor to the beginning of the next paragraph.

Positioning in the Window

<H>	Moves the cursor to the first line on the screen, or "home."
<M>	Moves the cursor to the middle line on the screen.
<L>	Moves the cursor to the last line on the screen.

Commands for Positioning in the File

Scrolling

<^f>	Scrolls the screen forward a full window, revealing the window of text below the current window.
<^d>	Scrolls the screen down a half window, revealing lines of text below the current window.
<^b>	Scrolls the screen back a full window, revealing the window of text above the current window.
<^u>	Scrolls the screen up a half window, revealing the lines of text above the current window.

Positioning on a Numbered Line

<G>	Moves the cursor to the beginning of the last line in the buffer.
<*n***G**>	Moves the cursor to the beginning of the *n*th line of the file (*n* = line number).

Searching for a Pattern

/*pattern*	Searchs forward in the buffer for the next occurrence of the *pattern* of text. Positions the cursor under the first character of the pattern.
?*pattern*	Searches backward in the buffer for the first occurrence of *pattern* of text. Positions the cursor under the first character of the pattern.

<n>	Repeats the last search command.
<N>	Repeats the search command in the opposite direction.

Commands for Inserting Text

<a>	Enters text input mode and appends text after the cursor.
<i>	Enters text input mode and inserts text before the cursor.
<o>	Enters text input mode by opening a new line immediately below the current line.
<O>	Enters text input mode by opening a new line immediately above the current line.
<ESC>	Escape; returns to command mode from text input mode (entered with any of the above commands).

Commands for Deleting Text

In Text Input Mode

<BACKSPACE>	Backspace; deletes the current character.
<^w>	Deletes the current word delimited by blanks.
<@>	Erases the current line of text.

In Command Mode

<x>	Deletes the current character.
<dw>	Deletes a word (or part of a word) from the cursor through the next space or to the next punctuation.
<dd>	Deletes the current line.
<*n***d***x*>	Deletes *n* number of text objects of type *x*, where *x* may be as a word, line, sentence, or paragraph.
<D>	Deletes the current line from the cursor to the end of the line.

Commands for Modifying Text

Characters, Words, Text Objects

<r>	Replaces the current character.
<s>	Deletes the current character and appends text until the **<ESC>** command is typed.
<S>	Replaces all the characters in the current line.
<~>	Changes upper case to lower case or lower case to upper case.
<cw>	Replaces the current word or the remaining characters in the current word with new text, from the cursor to the next space or punctuation.
<cc>	Replaces all the characters in the current line.
<*n***c***x*>	Replaces *n* number of text objects of type *x*, where *x* may be a word, line, sentence, or paragraph.

<**C**>	Replaces the remaining characters in the current line, from the cursor to the end of the line.

Cutting and Pasting Text

<**p**>	Places the contents of the temporary buffer (containing the output of the last delete or yank command) into the text after the cursor or below the current line.
<**yy**>	Yanks (extracts) a specified line of text and puts it into a temporary buffer.
<*n***y***x*>	Extracts a copy of *n* number of text objects of type *x* and puts it into a temporary buffer.
<"*l***y***x*>	Places a copy of text object *x* into a register named by a letter *l*. *x* may be a word, line, sentence, or paragraph.
<"*x***p**>	Places the contents of register *x* after the cursor or below the current line.

Other Commands

Special Commands

<^**g**>	Gives the line number of current cursor position in the buffer and modification status of the file.
<**.**>	Repeats the action performed by the last command.
<**u**>	Undoes the effects of the last command.
<**U**>	Restores the current line to its state prior to present changes.

<J>	Joins the line immediately below the current line with the current line.
<^l>	Clears and redraws the current window.

Line Editor Commands

:	Tells **vi** that the next commands you issue will be line editor commands.
:sh	Temporarily returns to the shell to perform some shell commands without leaving **vi**.
<^d>	Escapes the temporary return to the shell and returns to **vi** so you can edit the current window.
:*n*	Goes to the *n*th line of the buffer.
:*x*,*z***w** *filename*	Writes lines from the number *x* through the number *z* into a new file called *filename*.
:$	Moves the cursor to the beginning of the last line in the buffer.
:.,$d	Deletes all the lines from the current line to the last line.
:r *filename*	Inserts the contents of the file *filename* under the current line of the buffer.
:s/*text*/*new_text*/	Replaces the first instance of *text* on the current line with *new_text*.
:s/*text*/*new_text*/**g**	Replace every occurrence of *text* on the current line with *new_text*.
:g/*text*/**s**//*new_text*/**g**	Changes every occurrence of *text* in the buffer to *new_text*.

Commands for Quitting vi

<ZZ>	Writes the buffer to the file and quits **vi**.
:wq	Writes the buffer to the file and quits **vi**.
:w *filename* **:q**	Writes the buffer to the new file *filename* and quits **vi**.
:w! *filename* **:q**	Overwrites the existing file *filename* with the contents of the buffer and quits **vi**.
:q!	Quits **vi** whether or not changes made to the buffer were written to a file. Does not incorporate changes made to the buffer since the last write (**:w**) command.
:q	Quits **vi** if changes made to the buffer were written to a file.

Special Options for vi

vi *file1 file2 file3*	Enters three files into the **vi** buffer to be edited. Those files are *file1*, *file2*, and *file3*.
:w **:n**	When more than one file has been called on a single **vi** command line, writes the buffer to the file you are editing and then calls the next file in the buffer (use **:n** only after **:w**).
vi −**r** *file1*	Restores the changes made to *file1* that were lost because of an interrupt in the system.
view *file1*	Displays *file1* in the read-only mode of **vi**. Any changes made to the buffer will not be allowed to be written to the file.

Summary of Shell Command Language

This appendix is a summary of the shell command language and programming constructs discussed in Chapter 7, "Shell Tutorial." The first section reviews metacharacters, special characters, input and output redirection, variables and processes. These are arranged by topic in the order that they were discussed in the chapter. The second section contains models of the shell programming constructs.

The Vocabulary of Shell Command Language

Special Characters in the Shell

* ? [ ] .	Metacharacters; used to provide a shortcut to referencing file names, through pattern matching.
&	Executes commands in the background mode.
;	Sequentially executes several commands typed on one line, each pair separated by ;.
\	Turns off the meaning of the immediately following special character.
´...´	Enclosing single quotes turn off the special meaning of all characters.
"..."	Enclosing double quotes turn off the special meaning of all characters except $ and ` .

Redirecting Input and Output

<	Redirects the contents of a file into a command.
>	Redirects the output of a command into a new file, or replaces the contents of an existing file with the output.

> > Redirects the output of a command so that it is appended to the end of a file.

Directs the output of one command so that it becomes the input of the next command.

`*command*` Substitutes the output of the enclosed command in place of `*command*`.

Executing and Terminating Processes

batch Submits the following commands to be processed at a time when the system load is at an acceptable level. <^**d**> ends the **batch** command.

at Submits the following commands to be executed at a specified time. <^**d**> ends the **at** command.

at -l Reports which jobs are currently in the **at** or **batch** queue.

at -r Removes the **at** or **batch** job from the queue.

ps Reports the status of the shell processes.

kill *PID* Terminates the shell process with the specified process ID (PID).

nohup *command list* **&**
Continues background processes after logging off.

Making a File Accessible to the Shell

chmod ux *filename*
Gives the user permission to execute the file (useful for shell program files).

mv *filename* **$HOME/bin/***filename*
Moves your file to the **bin** directory in your home directory. This **bin** holds executable shell programs that you want to be accessible. Make sure the **PATH** variable in your **.profile** file specifies this **bin**. If it does, the shell will search in **$HOME/bin** for your file when you try to execute it. If your **PATH** variable does not include your **bin**, the shell

will not know where to find your file and your attempt to execute it will fail.

filename The name of a file that contains a shell program becomes the command that you type to run that shell program.

Variables

positional parameter
A numbered variable used within a shell program to reference values automatically assigned by the shell from the arguments of the command line invoking the shell program.

echo A command used to print the value of a variable on your terminal.

$# A special parameter that contains the number of arguments with which the shell program has been executed.

$* A special parameter that contains the values of all arguments with which the shell program has been executed.

named variable
A variable to which the user can give a name and assign values.

Variables Used in the System

HOME Denotes your home directory; the default variable for the **cd** command.

PATH Defines the path your login shell follows to find commands.

CDPATH Defines the search path for the **cd** command.

MAIL Gives the name of the file containing your electronic mail.

PS1 PS2	Define the primary and secondary prompt strings.
TERM	Defines the type of terminal.
LOGNAME	Login name of the user.
IFS	Defines the internal field separators (normally the space, the tab, and the carriage return).
TERMINFO	Allows you to request that the **curses** and **terminfo** subroutines search a specified directory tree before searching the default directory for your terminal type.
TZ	Sets and maintains the local time zone.

Shell Programming Constructs

Here Document

```
command <<!
input lines
!
```

For Loop

```
for variable<CR>
        in this list of values<CR>
do the following commands<CR>
        command 1<CR>
        command 2<CR>
            .<CR>
            .<CR>
        last command<CR>
done<CR>
```

While Loop

```
while command list<CR>
do<CR>
    command1<CR>
    command2<CR>
        .<CR>
        .<CR>
    last command<CR>
done<CR>
```

If...Then

```
if this command is successful<CR>
then command1<CR>
    command2<CR>
        .<CR>
        .<CR>
    last command<CR>
fi<CR>
```

If...Then...Else

```
if command list<CR>
    then command list<CR>
    else command list<CR>
fi<CR>
```

Case Construction

```
case word<CR>
in<CR>
    pattern1)<CR>
        command line 1<CR>
            .<CR>
            .<CR>
        last command line<CR>
    ;;<CR>
    pattern2)<CR>
        command line 1<CR>
            .<CR>
            .<CR>
        last command line<CR>
    ;;<CR>
    pattern3)<CR>
        command line 1<CR>
            .<CR>
            .<CR>
        last command line<CR>
    ;;<CR>
esac<CR>
```

break and continue Statements

A break or continue statement forces the program to leave any loop and execute the command following the end of the loop.

Setting the TERM Variable

AT&T supports many types of terminals for use with the UNIX system. Because some commands are terminal dependent, the system must know what type of terminal you are using whenever you log in. The system determines the characteristics of your terminal by checking the value of a variable called **TERM** which holds the name of a terminal. If you have put the name of your terminal into this variable, the system will be able to execute all programs in a way that is suitable for your terminal.

This method of telling the UNIX system what type of terminal you are using is called setting the terminal configuration. To set your terminal configuration, type the command lines shown on the following screen, substituting the name of your terminal for *terminal_name*.

```
$ TERM=terminal_name<CR>
$ export TERM<CR>
$ tput init<CR>
```

These lines must be executed in the order shown; otherwise, they will not work. Also, this procedure must be repeated every time you log in. Therefore, most users put these lines into a file called **.profile** that is automatically executed every time they log in. For details about the **.profile** file, see Chapter 7.

The first two lines in the screen tell the UNIX system shell what type of terminal you are using. The **tput init** command line instructs your terminal to behave in ways that the UNIX system expects a terminal of that type to behave. For example, it sets the terminal's left margin and tabs, if those capabilities exist for the terminal.

The **tput** command uses the entry in this database for your terminal to make terminal dependent capabilities and information available to the shell. Because the values of these capabilities differ for each type of terminal, you must execute the **tput init** command line every time you change the **TERM** variable.

For each terminal type, a set of capabilities is defined in a database. This database is usually found in either the **/usr/lib/terminfo** or **/usr/lib.COREterm** directory, depending on the system.

NOTE: Every system has at least one of these directories; some may have both. Your system administrator can tell you whether your system has the **terminfo** and/or the **.COREterm** directory.

The following sections describe how you can determine what *terminal_names* are acceptable. Further information about the capabilities in the **terminfo** database can be found on the **terminfo(4)** manual page in the *Programmer's Reference Manual*.

Acceptable Terminal Names

The UNIX system recognizes a wide range of terminal types. Before you put a terminal name into the **TERM** variable, you must make sure that your terminal is within that range.

You must also verify that the name you put into the **TERM** variable is a recognized terminal name. There are usually at least two recognized names: the name of the manufacturer and the model number. However, there are several ways to represent these names: by varying the use of uppercase and lowercase, using abbreviations, and so on. Do not put a terminal name in the **TERM** variable until you have verified that the system recognizes it.

The **tput** command provides a quick way to make sure your terminal is supported by your system. Type:

tput -T*terminal_name* **longname** <**CR**>

If your system supports your terminal it will respond with the complete name of your terminal. Otherwise, you will get an error message.

To find an acceptable name that you can put in the **TERM** variable, find a listing for your terminal in either of two directories: **/usr/lib/terminfo** or **/usr/lib/.COREterm**. Each of these directories is a collection of files with single-character names. Each file, in turn, holds a list of terminal names that all begin with the name of the file. (This name can be either a letter, such as the initial A in AT&T, or a number, such as the initial 5 in 5425.) Find the file whose name matches the first character of your terminal's name. Then list the file's contents and look for your terminal.

You can also check with your system administrator for a list of terminals supported by your system, and the acceptable names you can put in the **TERM** variable.

Example

Suppose your terminal is an AT&T Teletype Model 5425. Your login is **jim** and you are currently in your home directory. First, you verify that your system supports your terminal by running the **tput** command. Next, you find an acceptable name for it in the **/usr/lib/.COREterm/A** directory. The following screen shows which commands you need to do this:

```
$ tput -T5425 longname<CR>
AT&T 4425/5425
$ cd /usr/lib/.COREterm/A<CR>
$ ls
ATT4410
ATT4415
ATT4418
ATT4424
ATT4424-2
ATT4425
ATT4426
ATT513
ATT5410
ATT5418
ATT5420
ATT5420-2
ATT5425
ATT5620
ATT610BCT
ATTPT505
$
```

Now you are ready to put the name you found, `ATT5425`, in the **TERM** variable. Whenever you do this, you must also export **TERM** and execute **tput init**.

```
$ TERM=ATT5425<CR>
$ export TERM<CR>
$ tput init<CR>
$
```

The UNIX system now knows what type of terminal you are using and will execute commands appropriately.

GLOSSARY

Glossary

acoustic coupler
A device that permits transmission of data over an ordinary telephone line. When you place a telephone handset in the coupler, you link a computer at one end of the phone line to a peripheral device, such as a user terminal, at the other.

address
Generally, a number that indicates the location of information in the computer's memory. In the UNIX system, the address is part of an editor command that specifies a line number or range.

append mode
A text editing mode in which the characters you type are entered as text into the text editor's buffer. In this mode you enter (append) text after the current position in the buffer. See **text input mode**, compare with **command mode** and **insert mode**.

argument
The element of a command line that specifies data on which a command is to operate. Arguments follow the command name and can include numbers, letters, or text strings. For instance, in the command **lp –m myfile**, **lp** is the command and **myfile** is the argument. See **option**.

ASCII
(pronounced **as'-kee**) American Standard Code for Information Interchange, a standard for data transmission that is used in the UNIX system. ASCII assigns sets of 0s and 1s to represent 128 characters, including alphabetical characters, numerals, and standard special characters, such as #, $, %, and &.

AT&T 3B Computers
Computers manufactured by AT&T Technologies, Inc.

background
A type of program execution where you request the shell to run a command away from the interaction between you and the computer ("in the background"). While this

command runs, the shell prompts you to enter other commands through the terminal.

baud rate
A measure of the speed of data transfer from a computer to a peripheral device (such as a terminal) or from one device to another. Common baud rates are 300, 1200, 4800, and 9600. As a general guide, divide a baud rate by 10 to get the approximate number of English characters transmitted each second.

buffer
A temporary storage area of the computer used by text editors to make changes to a copy of an existing file. When you edit a file, its contents are read into a buffer, where you make changes to the text. For the changes to become a part of the permanent file, you must write the buffer contents back into the file. See **permanent file**.

child directory
See **subdirectory**.

command
The name of a file that contains a program that can be executed by the computer on request. Compiled programs and shell programs are forms of commands.

command file
See **executable file**.

command language interpreter
A program that acts as a direct interface between you and the computer. In the UNIX system, a program called the **shell** takes commands and translates them into a language understood by the computer.

command line
A line containing one or more commands, ended by typing a carriage return (**<CR>**). The line may also contain options and arguments for the commands. You type a command line to the shell to instruct the computer to perform one or more tasks.

command mode
A text editing mode in which the characters you type are interpreted as editing commands. This mode permits actions such as moving around in the buffer, deleting text, or moving lines of text. See **text input mode**, compare with **append mode** and **insert mode**.

context search
A technique for locating a specified pattern of characters (called a string) when in a text editor. Editing commands that cause a context search scan the buffer, looking for a match with the string specified in the command. See **string**.

control character
A nonprinting character that is entered by holding down the control key and typing a character. Control characters are often used for special purposes. For instance, when viewing a long file on your screen with the **cat** command, typing control-s (**^s**) stops the display so you can read it, and typing control-q (**^q**) continues the display.

current directory
The directory in which you are presently working. You have direct access to all files and subdirectories contained in your current directory. The shorthand notation for the current directory is a dot (.).

cursor
A cue printed on the terminal screen that indicates the position at which you enter or delete a character. It is usually a rectangle or a blinking underscore character.

default
An automatically assigned value or condition that exists unless you explicitly change it. For example, the shell prompt string has a default value of **$** unless you change it.

delimiter
A character that logically separates words or arguments on a command line. Two frequently used delimiters in the UNIX system are the space and the tab.

diagnostic
A message printed at your terminal to indicate an error encountered while trying to execute some command or program. Generally, you need not respond directly to a diagnostic message.

directory
A type of file used to group and organize other files or directories. You cannot directly enter text or other data into a directory. (For more detail, see Appendix A, Summary of the File System.)

disk
A magnetic data storage device consisting of several round plates similar to phonograph records. Disks store large amounts of data and allow quick access to any piece of data.

electronic mail
The feature of an operating system that allows computer users to exchange written messages via the computer. The UNIX system **mail** command provides electronic mail in which the addresses are the login names of users.

environment
The conditions under which you work while using the UNIX system. Your environment includes those things that personalize your login and allow you to interact in specific ways with the UNIX system and the computer. For example, your shell environment includes such things as your shell prompt string, specifics for backspace and erase characters, and commands for sending output from your terminal to the computer.

erase character
The character you type to delete the previous character you typed. The UNIX system default erase character is #; some users set the erase character to the BACKSPACE key.

escape
A means of getting into the shell from within a text editor or other program.

execute
The computer's action of running a program or command and performing the indicated operations.

executable file
A file that can be processed or executed by the computer without any further translation. When you type in the file name, the commands in the file are executed. See **shell procedure**.

file
A collection of information in the form of a stream of characters. Files may contain data, programs, or other text. You access UNIX system files by name. See **ordinary file**, **permanent file**, and **executable file**.

file name
A sequence of characters that denotes a file. (In the UNIX system, a slash character (/) cannot be used as part of a file name.)

file system
A collection of files and the structure that links them together. The UNIX file system is a hierarchical structure. (For more detail, see Appendix A, Summary of the File System.)

filter
A command that reads the standard input, acts on it in some way, and then prints the result as standard output.

final copy
The completed, printed version of a file of text.

foreground
The normal type of command execution. When executing a command in foreground, the shell waits for one command to end before prompting you for another command. In other words, you enter something into the computer and the computer "replies" before you enter something else.

full-duplex
A type of data communication in which a computer system can transmit and receive data simultaneously. Terminals and modems usually have settings for half-duplex (one-way) and full-duplex communication; the UNIX system uses the full-duplex setting.

full path name
A path name that originates at the root directory of the UNIX system and leads to a specific file or directory. Each file and directory in the UNIX system has a unique full path name, sometimes called an absolute path name. See **path name**.

global
A term that indicates the complete or entire file. While normal editor commands commonly act on only the first instance of a pattern in the file, global commands can perform the action on all instances in the file.

hardware
The physical machinery of a computer and any associated devices.

hidden character
One of a group of characters within the standard ASCII character set that are not printable. Characters such as backspace, escape, and <^d> are examples.

home directory
The directory in which you are located when you log in to the UNIX system; also known as your login directory.

input/output
The path by which information enters a computer system (input) and leaves the system (output). An input device that you use is the terminal keyboard and an output device is the terminal display.

insert mode
A text editing mode in which the characters you type are entered as text into the text editor's buffer. In this mode you enter (insert) text before the current position in the buffer. See **text input mode**, compare with **append**

mode and **command mode**.

interactive
Describes an operating system (such as the UNIX system) that can handle immediate-response communication between you and the computer. In other words, you interact with the computer from moment to moment.

line editor
An editing program in which text is operated upon on a line-by-line basis within a file. Commands for creating, changing, and removing text use line addresses to determine where in the file the changes are made. Changes can be viewed after they are made by displaying the lines changed. See **text editor**, compare with **screen editor**.

login
The procedure used to gain access to the UNIX operating system.

login directory
See **home directory**.

login name
A string of characters used to identify a user. Your login name is different from other login names.

log off
The procedure used to exit from the UNIX operating system.

metacharacter
A subset of the set of special characters that have special meaning to the shell. The metacharacters are *****, **?**, and the pair **[ ]**. Metacharacters are used in patterns to match file names.

mode
In general, a particular type of operation (for example, an editor's append mode). In relation to the file system, a mode is an octal number used to determine who can have access to your files and what kind of access they can have. See **permissions**.

modem
A device that connects a terminal and a computer by way of a telephone line. A modem converts digital signals to tones and converts tones back to digital signals, allowing a terminal and a computer to exchange data over standard telephone lines.

multitasking
The ability of an operating system to execute more than one program at a time.

multiuser
The ability of an operating system to support several users on the system at the same time.

nroff
A text formatter available as an add-on to the UNIX system. You can use the **nroff** program to produce a formatted on-line copy or a printed copy of a file. See **text formatter**.

operating system
The software system on a computer under which all other software runs. The UNIX system is an operating system.

option
Special instructions that modify how a command runs. Options are a type of argument that follow a command and usually precede other arguments on the command line. By convention, an option is preceded by a minus sign (−); this distinguishes it from other arguments. You can specify more than one option for some commands given in the UNIX system. For example, in the command **ls −l −a directory**, **−l** and **−a** are options that modify the **ls** command. See **argument**.

ordinary file
A file, containing text or data, that is not executable. See **executable file**.

output
Information processed in some fashion by a computer and delivered to you by way of a printer, a terminal, or a

similar device.

parameter
A special type of variable used within shell programs to access values related to the arguments on the command line or the environment in which the program is executed. See **positional parameter**.

parent directory
The directory immediately above a subdirectory or file in the file system organization. The shorthand notation for the parent directory is two dots (..).

parity
A method used by a computer for checking that the data received matches the data sent.

password
A code word known only to you that is called for in the login process. The computer uses the password to verify that you may indeed use the system.

path name
A sequence of directory names separated by the slash character (/) and ending with the name of a file or directory. The path name defines the connection path between some directory and the named file.

peripheral device
Auxiliary devices under the control of the main computer, used mostly for input, output, and storage functions. Some examples include terminals, printers, and disk drives.

permanent file
The data stored permanently in the file system structure. To change a permanent file, you can make use of a text editor, which maintains a temporary work space, or buffer, apart from the permanent files. Once changes have been made to the buffer, they must be written to the permanent file to make the changes permanent. See **buffer**.

permissions
Access modes, associated with directories and files, that permit or deny system users the ability to read, write, and/or execute the directories and files. You determine the permissions for your directories and files by changing the mode for each one with the **chmod** command.

pipe
A method of redirecting the output of one command to be the input of another command. It is named for the character ¦ that redirects the output. For example, the shell command **who ¦ wc −l** pipes output from the **who** command to the **wc** command, telling you the total number of people logged into your UNIX system.

pipeline
A series of filters separated by ¦ (the pipe character). The output of each filter becomes the input of the next filter in the line. The last filter in the pipeline writes to its standard output, or may be redirected to a file. See **filter**.

positional parameters
Numbered variables used within a shell procedure to access the strings specified as arguments on the command line invoking the shell procedure. The name of the shell procedure is positional parameter **$0**. See **variable** and **shell procedure**.

prompt
A cue displayed at your terminal by the shell, telling you that the shell is ready to accept your next request. The prompt can be a character or a series of characters. The UNIX system default prompt is the dollar sign character ($).

printer
An output device that prints the data it receives from the computer on paper.

process
Generally a program that is at some stage of execution. In the UNIX system, it also refers to the execution of a computer environment, including contents of memory,

register values, name of the current directory, status of files, information recorded at login time, and various other items.

program
The instructions given to a computer on how to do a specific task. Programs are user-executable software.

read-ahead capability
The ability of the UNIX system to read and interpret your input while sending output information to your terminal in response to previous input. The UNIX system separates input from output and processes each correctly.

relative path name
The path name to a file or directory which varies in relation to the directory in which you are currently working.

remote system
A system other than the one on which you are working.

root
The source directory of all files and directories in the file system; designated by the slash character (/).

screen editor
An editing program in which text is operated on relative to the position of the cursor on a visual display. Commands for entering, changing, and removing text involve moving the cursor to the area to be altered and performing the necessary operation. Changes are viewed on the terminal display as they are made. See **text editor**, compare with **line editor**.

search pattern
See **string**.

search string
See **string**.

secondary prompt
A cue displayed at your terminal by the shell to tell you that the command typed in response to the primary prompt is incomplete. The UNIX system default secondary prompt is the "greater than" character (>).

shell
A UNIX system program that handles the communication between you and the computer. The shell is also known as a command language interpreter because it translates your commands into a language understandable by the computer. The shell accepts commands and causes the appropriate program to be executed.

shell procedure
An executable file that is not a compiled program. A shell procedure calls the shell to read and execute commands contained in a file. This lets you store a sequence of commands in a file for repeated use. It is also called a shell program or command file. See **executable file**.

silent character
See **hidden character**.

software
Instructions and programs that tell the computer what to do. Contrast with **hardware**.

source code
The uncompiled version of a program written in a language such as C or Pascal. The source code must be translated to machine language by a program known as a compiler before the computer can execute the program.

special character
A character having special meaning to the shell program and used for common shell functions such as file redirection, piping, background execution, and file name expansion. The special characters include <, >, |, ;, &, *, ?, [, and].

special file
A file (called a device driver) used as an interface to an input/output device, such as a user terminal, a disk drive, or a line printer.

standard input
An open file that is normally connected directly to the keyboard. Standard input to a command normally goes

from the keyboard to this file and then into the shell. You can redirect the standard input to come from another file instead of from the keyboard; use an argument in the form < **file**. Input to the command will then come from the specified file.

standard output
An open file that is normally connected directly to a primary output device, such as a terminal printer or screen. Standard output from the computer normally goes to this file and then to the output device. You can redirect the standard output into another file instead of to the printer or screen; use an argument in the form > **file**. Output will then go to the specified file.

string
Designation for a particular group or pattern of characters, such as a word or phrase, that may contain special characters. In a text editor, a context search interprets the special characters and attempts to match the specified pattern with a string in the editor buffer.

string variable
A sequence of characters that can be the value of a shell variable. See **variable**.

subdirectory
A directory pointed to by a directory one level above it in the file system organization; also called a child directory.

system administrator
The person who monitors and controls the computer on which your UNIX system runs; sometimes referred to as a super-user.

terminal
An input/output device connected to a computer system, usually consisting of a keyboard with a video display or a printer. A terminal allows you to give the computer instructions and to receive information in response.

text editor
Software for creating, changing, or removing text with the aid of a computer. Most text editors have two modes--an input mode for typing in text and a command

mode for moving or modifying text. Two examples are the UNIX system editors **ed** and **vi**. See **line editor** and **screen editor**.

text formatter
A program that prepares a file of text for printed output. To make use of a text formatter, your file must also contain some special commands for structuring the final copy. These special commands tell the formatter to justify margins, start new paragraphs, set up lists and tables, place figures, and so on. Two text formatters available as add-ons to your UNIX system are **nroff** and **troff**.

text input mode
A text editing mode in which the characters you type are entered as text into the text editor's buffer. To execute a command, you must leave text input mode. See **command mode**, compare with **append mode** and insert mode.

timesharing
A method of operation in which several users share a common computer system seemingly simultaneously. The computer interacts with each user in sequence, but the high-speed operation makes it seem that the computer is giving each user its complete attention.

tool
A package of software programs.

troff
A text formatter available as an add-on to the UNIX system. The **troff** program drives a phototypesetter to produce high-quality printed text from a file. See **text formatter**.

tty
Historically, the abbreviation for a teletype terminal. Today, it is generally used to denote a user terminal.

user
Anyone who uses a computer or an operating system.

user-defined
Something determined by the user.

user-defined variable
A named variable given a value by the user. See **variable**.

UNIX system
A general-purpose, multiuser, interactive, time-sharing operating system developed by AT&T Bell Laboratories. The UNIX system allows limited computer resources to be shared by several users and efficiently organizes the user's interface to a computer system.

utility
Software used to carry out routine functions or to assist a programmer or system user in establishing routine tasks.

variable
A symbol whose value may change. In the shell, a variable is a symbol representing some string of characters (a **string value**). Variables may be used in an interactive shell as well as within a shell procedure. Within a shell procedure, positional parameters and keyword parameters are two forms of variables. (Keyword parameters are discussed fully in "Shell Commands and Programming".)

video display terminal
A terminal that uses a television-like screen (a monitor) to display information. A video display terminal can display information much faster than printing terminals.

visual editor
See **screen editor**.

working directory
See **current directory**.

INDEX

Index

absolute path name ... **3**:8
acceptable terminal names ... **F**:2-4
active computer ... **8**:56
alias ... **8**:17, 19, 27, 30, 43, 45
background mode ... **7**:10-11, 20, 37, 103; **B**:1; **E**:1; **F**:6
backslash ... **2**:10; **5**:47, 65-66, 72; **7**:12
banner ... **7**:13-14, 21, 26, 29, 37, 100; **8**:35; **B**:1
BASIC ... **4**:13; **A**:3
Basic Windowing Utilities ... **F**:6
baud rate ... **2**:3, 15, 20; **8**:70
calling another unix system ... **2**:15; **8**:69
case construction ... **7**:83-85; **E**:7
cat ... **3**:21, 27, 33-34, 36, 47, 72-73; **4**:7-8; **6**:86; **7**:17, 37-39, 44, 48, 55, 67-68, 102; **8**:72; **A**:3; **B**:1
cd ... **3**:16, 26-27, 60, 65; **4**:8; **7**:12, 37, 41, 53, 96-97, 103; **8**:50; **A**:2; **B**:2-3; **E**:3
change text ... **1**:2, 9; **3**:67-68; **4**:3-5; **5**:1, 4, 10, 12, 16-17, 32, 34, 39, 45, 50, 52-53, 65-66, 69, 81, 88, 93; **6**:1, 65, 71, 75, 83; **7**:66; **8**:20, 27; **D**:8
changing directories ... **3**:26, 60, 64-65
changing permissions ... **3**:60, 63-65; **7**:40, 96; **8**:49-50
character counts ... **3**:57-58; **5**:14; **7**:21, 66; **B**:5
child directory ... **3**:12, 19, 26
chmod ... **3**:23, 60, 63-66; **4**:10; **7**:40, 44, 55, 96; **8**:49-50; **E**:2
command line syntax ... **1**:10-11; **8**:48, 51
command mode ... **1**:3; **3**:22, 60, 67; **4**:3-4; **5**:4-7, 9, 37, 39-40, 44-45, 47, 84, 86; **6**:6, 11-12, 22, 60, 64, 66, 75, 84, 86; **7**:10-11, 37, 103; **8**:17-18, 20-21, 23, 26-27, 31, 33, 35, 41, 45-46, 66-67, 69, 72; **B**:1; **C**:1, 3; **D**:2, 5-6; **E**:1; **F**:6
configuration checks ... **2**:3, 20
connecting remote terminal ... **8**:66-68, 70
continue statements ... **7**:87; **E**:7
control characters ... **2**:11-12, 25; **5**:87
copy-to list ... **8**:27
COREterm ... **F**:2
cp ... **3**:50-52, 68; **7**:93; **B**:2
ct ... **8**:1, 66-68
cu command ... **8**:66, 69-70, 73
current directory ... **3**:6, 11, 13, 16, 18-21, 26, 50, 54, 57-58, 65; **5**:73; **7**:4-7, 10, 12, 16, 27, 35, 39-40, 80-82, 97, 100, 102, 107; **8**:13-14, 30, 38, 52-53, 64, 73, 75; **B**:2, 4
data file ... **1**:1, 8, 11; **4**:8; **7**:43; **8**:51, 55, 70; **A**:3
date ... **2**:9, 22-23; **3**:18, 43; **4**:9-10; **5**:90; **7**:21, 23-24, 26-29, 37, 58, 89, 100, 105;

8:12, 34, 60; **A:**3; **B:**1-2
default environment ... **4:**7-8; **5:**47; **7:**93
defined aliases ... **8:**45
defined variables ... **7:**53; **8:**44
delete functions ... **2:**8-10; **5:**47
deleting mail ... **8:**13, 32-33, 37, 41; **B:**3
deleting text ... **1:**5; **2:**8, 10; **4:**2, 4; **5:**1, 3, 5, 7-9, 16-17, 32, 44-48, 73, 75; **6:**1, 3, 6, 71, 75, 78, 100; **C:**3-4; **D:**6-7
destination directory ... **3:**13; **8:**48-49
destination-file ... **8:**51-54
diff ... **3:**67-68; **B:**2
directory ... **1:**5-8, 11; **3:**1-2, 4, 6-8, 11-16, 18-23, 26-29, 31, 33, 50-51, 53-58, 60-61, 65; **4:**7-8, 12; **5:**31, 73, 79, 93; **6:**5, 70, 95; **7:**4-8, 10, 12, 16, 27, 35, 37, 39-42, 51, 53, 68, 70, 80-82, 93, 96-97, 100, 102, 107; **8:**6, 13-14, 17-18, 30-31, 38, 44-45, 48-58, 60, 62-64, 73, 75; **A:**1-3; **B:**2-5; **E:**2-4; **F:**2, 4
display commands ... **3:**33-42
echo ... **2:**16; **7:**4, 37, 39, 44, 49-51, 53-55, 60, 68, 70, 74, 76-77, 87, 93-94, 96-97, 100, 102, 105, 108; **8:**72; **B:**2; **E:**3
editing messages ... **5:**83, 85; **6:**94; **8:**17, 20, 22
else ... **7:**77, 85; **E:**6
environment ... **1:**1, 9-10; **3:**60; **4:**2, 7-8, 13; **5:**47; **6:**3, 5; **7:**1, 53, 86, 93-94, 101, 108; **8:**16-18, 24, 28, 43-44, 46
erase function ... **2:**8-10; **4:**8; **5:**47
executing commands, remote system ... **8:**1, 74
exit ... **2:**25; **5:**5; **6:**87, 94; **7:**62, 68-69, 85; **8:**23, 26, 31, 68
exiting mailx ... **8:**26, 31
file system ... **1:**1, 3-11; **2:**16, 18, 23; **3:**1-2, 4, 6-8, 11, 13, 15-16, 18, 20, 26, 31, 33, 39, 43, 48, 50-51, 53, 55, 57, 60-61, 64-67, 69, 72, 75; **4:**2, 12; **5:**3, 10, 83, 90; **6:**5, 91-92; **7:**1, 4, 6-8, 15, 42, 75, 77, 90, 93; **8:**1, 19, 25, 34, 42, 47-49, 51-52, 54-55, 57-58, 62-64, 69-70, 74-75;1; **A:**1-3; **B:**3-5; **D:**9; **F:**6
flow of control ... **1:**8, 12; **3:**45
folder variable ... **8:**44-45
full path name ... **3:**6-8, 14, 16, 19-20, 26-27, 51-52, 54; **7:**12, 43; **8:**14, 51-52, 54, 75; **B:**4
glossary ... **2:**24; **C:**1; **D:**1
grep ... **3:**67, 69-71; **7:**11-13, 20, 27, 32, 34-35, 46-47, 54, 76-77, 92; **8:**9; **A:**3; **B:**2
help ... **2:***3-25;* **5:**83-86; **C:**4; **F:**6
HOME ... **E:**3
home directory ... **1:**6; **3:**4, 6, 16, 21, 26, 29; **4:**8; **5:**31; **6:**5; **7:**53, 81, 93, 97; **8:**13, 17-18, 31, 38, 44, 49-53; **A:**1; **B:**2; **E:**2-3; **F:**4

if-endif command ... **8:**45
incorporate existing text ... **8:**20, 24
input redirection ... **3:**36, 43; **7:**10, 15, 20, 63; **8:**47; **E:**1
job number ... **3:**48; **7:**27, 29-30, 103; **8:**58, 60; **B:**1, 4
job queue ... **7:**27, 29; **8:**58, 60; **B:**4; **E:**2
kernel ... **1:**1, 3-4, 6-7, 12-13
kill function ... **2:**8-9; **4:**8; **5:**47
layers ... **3:**2, 11; **F:**6-7, 10
leaving mailx ... **8:**13, 31, 40-41
lex ... **4:**13; **B:**2
libwindows ... **F:**10
Line Addressing ... **5:**6, 16
line addressing commands ... **5:**9, 15-18, 24-28, 31-32, 34-35, 37, 39, 50-53, 59, 73, 75; **C:**1-3
line kill function ... **2:**8-9; **5:**47
link count ... **3:**23
linking computers ... **1:**1; **2:**17; **8:**55, 66, 69
local system ... **2:**13; **8:**3, 8, 47, 52, 58
lock enabling ... **2:**20; **3:**62
log off ... **2:**1, 9, 12, 25; **4:**7; **5:**93; **7:**11, 35; **8:**66, 70; **B:**3
login name ... **2:**1, 10, 12-13, 16, 18, 23; **3:**4, 6, 23; **4:**8, 12; **7:**15, 45, 53, 96; **8:**3-4, 7-9, 12, 17, 19-20, 27, 30, 34, 38, 45, 47, 51-52, 58, 60; **B:**3, 5; **E:**3
login procedure ... **2:**16-19; **3:**4; **7:**21
LOGNAME ... **7:**53; **E:**3
loop ... **7:**1, 42-43, 62, 69-70, 72-75, 80, 87-88, 100; **E:**4-5, 7
lp ... **3:**33, 43, 47-48; **8:**75; **A:**3; **B:**2
lpstat ... **3:**48; **B:**2
ls ... **3:**16, 18-24, 50, 53-54, 56-58, 60-62, 64-65; **4:**7; **5:**50, 83, 86-87, 93; **6:**11, 84; **7:**4-8, 12, 16, 37, 39-43, 93, 102; **8:**13, 50, 70; **A:**2; **B:**3; **C:**3-4; **D:**2, 4, 7-8
mail ... **1:**8, 10; **3:**48; **4:**8, 12; **5:**93; **7:**15, 21, 29-30, 45, 53, 63-64, 89-91; **8:**1-9, 12-14, 16-18, 20, 26-27, 32-33, 36-42, 44-45, 47-48, 53-54, 58-59, 62-63, 74; **A:**3; **B:**3; **E:**3
mailbox ... **8:**12-14, 19, 26, 33, 35-38, 41
mailrc ... **8:**16-19, 23, 42, 44-46
mailx ... **1:**8, 10; **3:**48; **4:**8, 12; **5:**93; **7:**15, 21, 29-30, 45, 53, 63-64, 89-91; **8:**1-9, 12-14, 16-24, 26-28, 31-42, 44-48, 53-54, 58-59, 62-63, 74; **A:**3; **B:**3; **E:**3
managing incoming mail ... **8:**5-6, 12, 16-18, 27, 32
merging files ... **3:**67, 72-73
mkdir ... **3:**16-17, 26; **7:**41, 96; **A:**3; **B:**3
msglist ... **8:**32, 34-35, 37-38, 42
mv ... **3:**53-54; **7:**8, 41-42, 71-72, 80-82, 100, 107; **B:**3; **E:**2
networking ... **8:**1, 54, 56, 66

octal method ... **3**:65
ordinary file ... **1**:5-6; **3**:2, 23
outfolder variable ... **8**:44-45
output redirection ... **3**:36, 43; **7**:10, 15, 17-20; **E**:1
parent directory ... **3**:11-13, 19, 23, 28-29
password ... **2**:1, 10, 16-19; **8**:73
PATH ... **E**:2-3
path names ... **3**:6-8, 11-16, 18-20, 26-27, 51-54, 60; **7**:12, 40, 42-43, 70, 81, 96-97; **8**:14, 34, 48, 51-55, 63, 75; **A**:2; **B**:2, 4
permission ... **1**:6; **3**:1, 4, 20, 23, 28, 60-65; **4**:10; **7**:40-41, 68, 80, 96; **8**:36, 48-51, 55; **E**:2
pg ... **3**:21, 33, 37, 39-41; **6**:86; **B**:3
pr ... **3**:21, 33, 42-45, 47; **7**:6, 9; **8**:75; **B**:4
print command ... **3**:43-49
profile ... **2**:9-10; **3**:21, 41; **6**:5; **7**:93-94, 96-99, 108; **8**:23; **E**:2
protecting files ... **3**:60, 63
public directory ... **4**:12; **8**:49, 53, 57-58, 60, 62-64; **B**:5
put command ... **1**:10; **3**:69; **5**:4, 8, 39, 79; **6**:88; **7**:27, 98; **8**:20, 27, 45, 73
pwd ... **3**:6-8, 18, 26; **7**:12, 37, 39, 42, 96; **B**:4
quitting vi ... **D**:2, 9
read permission ... **1**:6; **3**:23, 60-61, 63-65; **7**:68; **8**:49-50
reading mail ... **4**:12; **7**:64, 89; **8**:2, 6-7, 12-14, 17-18, 26, 32-33, 41, 45, 47
reassigning delete functions ... **2**:8-10
receiving files ... **1**:6; **3**:33, 50, 53; **4**:12; **5**:4, 8; **6**:92; **7**:15-16, 19; **8**:1-2, 6, 47, 55, 62, 70
record variable ... **8**:44-45
redirecting input ... **4**:7; **7**:1-2, 15, 20, 62-63, 90; **8**:47; **E**:1
relative path name ... **3**:6-7, 11-14, 16, 19-20, 26-27, 51-52, 54; **8**:14, 52
remote command execution ... **8**:74
remote system ... **4**:12-13; **8**:1, 3, 8-10, 47-48, 52-53, 55, 58, 66, 69, 74; **B**:5
removing file ... **1**:5-6, 10; **3**:28-29, 55-56, 65; **5**:7, 91; **6**:101; **7**:20-21; **8**:17; **B**:4
renaming file ... **3**:53-54; **5**:10, 90; **7**:8
rm ... **3**:55-56; **7**:5, 21; **B**:4
rmdir ... **3**:16, 28-29; **B**:4
root directory ... **1**:6-7; **3**:2, 6-8, 11; **A**:1-3
saving mail ... **8**:14, 32, 38, 41; **B**:3
scrolling ... **4**:5; **6**:1, 26-27, 95; **7**:18; **D**:4
set command ... **1**:9; **3**:68; **4**:2, 5, 8; **5**:73; **6**:4; **7**:43-44, 54, 62, 70, 77, 85, 87, 93, 98; **8**:16, 43-46, 72; **A**:2; **B**:1; **F**:1, 6
shell ... **1**:1, 3, 8-13; **2**:9-10, 24-25; **3**:75; **4**:1-4, 7-9, 11; **5**:1, 11-12, 14, 19, 44, 47-48, 79, 83, 86, 89-90, 93; **6**:4, 6, 86-87, 91; **7**:1-2, 9-13, 15-16, 19-20, 26-27, 30,

32-33, 36, 38-49, 51-54, 56, 58, 61-63, 65, 68-72, 75-76, 80-82,

85-86, 88-89, 92-94, 96-98, 100, 103; **8:**24, 31, 33, 42, 44-45, 75; **B:**1; **C:**5; **D:**1, 8; **E:**1-4; **F:**1-2, 6, 8-10
shell programming constructs ... **7:**1, 62, 69-70, 75-76; **E:**1, 4
shell script ... **1:**9; **4:**9, 11; **7:**65; **B:**1
software development, tools ... **1:**2, 9; **4:**13
sort command ... **3:**72-73; **7:**18-19
source-file ... **8:**51-53
special character ... **2:**6, 10, 18; **3:**70; **5:**1, 60, 65, 69, 90, 104, 108; **7:**1-2, 4, 10, 12-13, 20, 32, 74; **8:**36, 75; **C:**3-4; **E:**1
specification file ... **8:**19; **F:**7-10
spell command ... **7:**18
spool directory ... **8:**51, 55-56
status report ... **8:**54, 59-60; **B:**5; **E:**2
symbolic method ... **3:**65
take command ... **4:**8; **5:**4, 31-32, 35, 45; **6:**73; **7:**10, 18, 20; **8:**1, 20, 32, 44, 73; **B:**5
TERM ... **3:**41; **E:**3; **F:**1-4, 9
terminal configuration ... **2:**3, 20; **6:**3-4, 6; **F:**1
tilde escapes ... **8:**17-18, 20-23, 25, 27-28, 31
time command ... **2:**16, 22; **3:**7, 22, 33, 37, 39, 43, 55; **4:**4, 8-9; **5:**84; **6:**5, 86, 95; **7:**2, 10, 18, 23-24, 26-30, 32, 37, 42-43, 47, 58, 60, 69, 71, 88, 98, 100; **8:**14, 49, 51, 64; **B:**1; **E:**2; **F:**2, 6
tput command ... **6:**5; **7:**98; **D:**1; **F:**1-2, 4
uname ... **8:**1, 8-10; **B:**5
undeliverable mail ... **8:**4
uucico ... **8:**51, 55-56
uucp ... **4:**12; **7:**96; **8:**1-2, 47-56, 59; **B:**5
uuname ... **8:**1, 8-10, 70; **B:**5
uupick ... **4:**12; **8:**1, 58, 62-65; **B:**5
uustat ... **8:**1, 58-61; **B:**5
uuto ... **4:**12; **8:**1-2, 47-50, 57-62; **B:**5
uux ... **4:**12; **8:**1, 66, 74-75
wc ... **1:**11; **3:**57-59; **4:**9-10; **B:**5
who command ... **2:**22-23; **3:**23, 60, 63, 65; **4:**9; **7:**46, 75
windowing ... **5:**14; **6:**1, 6, 34, 41, 43-44, 48, 56, 84; **D:**2, 4, 8
work file ... **2:**9, 23; **3:**1, 4, 6, 8, 11, 18, 22, 29, 31, 44, 60, 66; **4:**2-4, 12; **5:**1, 4-5, 10, 12, 73, 88; **6:**79; **7:**12, 20, 40; **8:**36, 51-52, 54-56, 58; **F:**6
write permission ... **1:**6; **3:**28, 60-61, 63-65; **4:**10; **7:**80; **8:**48-49, 51
yacc ... **4:**13; **B:**5